



AD-A145 522

DTIC FILE COPY

AD

FUNDING PROJECT NO. IT665702D625

TECOM PROJECT (TRMS) NO. 7-CO-RDO-EP1-004

TEST ACTIVITY REPORT NO. USAEPG-FR-1263

TEST SPONSOR: US ARMY TEST AND  
EVALUATION COMMAND

METHODOLOGY INVESTIGATION

FINAL REPORT

PROGRAM FLOW ANALYZER

VOLUME III

BY  
LESLIE F. CLAUDIO

AUGUST 1984

DISTRIBUTION UNLIMITED

20030109002

US ARMY ELECTRONIC PROVING GROUND  
Fort Huachuca, Arizona 85613-7110

DTIC  
ELECTE  
SEP 12 1984  
S E D

84 09

07 090  
10 042

# APPENDIX H

## SOFTWARE MODULE COMPLEXITY METRICS

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



## TABLE OF CONTENTS

	PAGE
1.0 INTRODUCTION .....	1
1.1 Software Complexity .....	2
2.0 USES OF SOFTWARE METRICS .....	3
3.0 HALSTEAD'S SOFTWARE SCIENCE .....	4
3.1 List of Halstead's Metrics .....	7
3.2 Description of Operators and Operands .....	8
3.3 Program Difficulty Used to Support Testability .....	10
3.4 Overall Evaluation of Software Science .....	11
4.0 MCCABE'S SOFTWARE COMPLEXITY METRICS .....	12
4.1 Cyclomatic Complexity .....	12
4.2 Essential Complexity .....	13
4.3 Actual Complexity .....	15
5.0 KNOTS COMPLEXITY MEASURE .....	15
5.1 Knots Complexity Using Flow Graphs .....	17
5.2 Knots and Cyclomatic Complexity .....	18
5.3 Knots Complexity and Structured Programming .....	21
6.0 GILB'S SOFTWARE METRICS .....	24
6.1 Reliability Metrics .....	24
6.2 Flexibility Metrics .....	26
6.3 Structure Metrics .....	27
6.4 Performance Metrics .....	28
6.5 Resource Metrics .....	28
6.6 Diverse Metrics .....	29
6.7 Summary of Gilb's Metrics .....	29
7.0 BOEHM'S QUANTITATIVE EVALUATION OF S/W QUALITY .....	30
8.0 THAYER'S SOFTWARE RELIABILITY .....	36
9.0 MYER'S EXTENSION TO MCCABE'S CYCLOMATIC COMPLEXITY .....	39
10.0 COMPARISON OF HALSTEAD'S, MCCABE'S, AND KNOTS METRICS .....	41
11.0 CONCLUSIONS AND RECOMMENDATIONS .....	46
12.0 REFERENCES .....	47

## 1.0

## INTRODUCTION

Quantitative techniques for describing and evaluating software systems have been developed on a largely ad hoc basis. The need for control over the software development process has created a software engineering discipline whose purpose is to establish operational procedures for the development of quality software. Unfortunately, techniques for predicting and evaluating software quality and performance are just emerging for practical application. Computer science and software engineering need metrics to help quantify the various aspects of software development. This report summarizes and explains some metrics from current literature.

The emphasis of this report is the review of software metrics as they may be applied to the maintainability, supportability, and testability of software. In particular, the metrics suggested by Halstead, McCabe, and Woodward et al, are reviewed. However, metrics of a more general nature are presented for completeness, consideration, and as a historical background. The following software metrics are summarized.

- 1) Halstead's Software Sciences
- 2) McCabe's Complexity Metrics
- 3) Woodward, Hennen, and Hedley's Knots Complexity
- 4) Gilb's Software Metrics
- 5) Boehm, Lipow, and Brown's Software Metrics
- 6) Thayer's Reliability Metrics
- 7) McCall's Software Metric Approach (see Section on Boehm)
- 8) Myers' Extension to McCabe's Complexity

The emphasis for software metrics discussed here is placed on module complexity. Recommendations are included for the near term use of metrics for software assessment. A subsequent report will address module strength, cohesion, and inter-connectivity.

#### 1.1 Software Complexity

The measurement of software complexity has received increased attention, since software costs have increased in proportion to total computer system costs. Heretofore, complexity has been a loosely defined term, and neither Boehm or McCall included it directly among their metrics for software quality.

Two separate focuses have emerged in studying software complexity: computational and psychological complexity. Computational complexity relies on the formal mathematical analysis of such problems as algorithm efficiency and use of machine resources. In contrast to this formal analysis, the empirical study of psychological complexity has emerged from the understanding that software development and maintenance are largely human activities. Psychological complexity is concerned with the characteristics of software which affect programmer performance.

## 2.0 USES OF SOFTWARE METRICS

The measurement of software complexity is one facet of a larger effort to measure important software characteristics. Measurements of software characteristics can provide valuable information throughout the software life cycle. During development, measurements can be used to predict the resources which will be required in future phases of a project. For instance, metrics determined from the detailed design can be used to predict the amount of effort that will be required to implement and test the code. Metrics determined from the code can be used to predict the number of errors that may be found in subsequent testing or the difficulty involved in the maintainability and supportability of a section of code. Metrics should be utilized during software testing to assess the complexity and the quality of software.

Software metrics can be used in at least three ways during the software lifecycle:

### Management Information Tools

As a management tool, metrics provide several types of information for prediction and assessment. Measurements can be developed for costing and sizing at the project level and for estimating productivity. Such metrics allow managers to assess progress, future problems, and resource requirements. If these metrics can be proven reliable and valid indicators of development processes, they provide an excellent source of management vision into a software project.

### Measures of Software Quality

Interest grows in creating quantifiable criteria against which a software product can be judged. An example criteria would be the minimally acceptable mean time between software failures or the time required to modify a module. These criteria could be used as either acceptance standards by a software acquisition manager or as guidance

acceptance standards by a software acquisition manager or as guidance to potential problems in the code during software validation and verification.

#### Feedback to Software Personnel

Software complexity metrics may be used to provide feedback to programmers about their code. When a section of software grows too complex, the code may be redesigned until metric values are brought within acceptable limits or the complexity is justified.

Software metrics may be differentiated between measures of process and measures of product. Measures of process would include the resource estimation metrics described as potential management tools. However, measures of process convey little information about the actual state of the software product. Measures of the product represent software characteristics as they exist at a given time, but do not indicate how the software has evolved into this state. Measures used for feedback to programmers or to assess quality of software are measures of product. The latter metrics are of primary concern to software assessment efforts.

### 3.0 HALSTEAD'S SOFTWARE SCIENCE

The theory of software science as presented by Halstead in his 1977 book, Elements of Software Science[1], is summarized here. Halstead's theory is probably the best known and the most thoroughly studied measures of software complexity. A computer program is considered in software science to be a series of tokens which can be classified as either "operators" or "operands". All software science measures are functions of the counts of these tokens. The basic units are defined as

$n_1$  = number of unique operators  
 $n_2$  = number of unique operands  
 $N_1$  = total occurrences of operators  
 $N_2$  = total occurrences of operands.

Generally, any symbol or keyword in a program that specifies an algorithmic action is considered an operator, while a symbol used to represent data is considered an operand. Most punctuation marks are also categorized as operators. This is illustrated in the example presented below in Section 3.2. The size of the vocabulary of a program, which consists of the number of unique tokens used to build a program, is defined as:

$$n = n_1 + n_2.$$

The length of the program in terms of the total number of tokens used is:

$$N = N_1 + N_2.$$

It should be noted that  $N$  is closely related to the traditional "lines of code" (LOC) measure of program length. For machine language programs where each line consists of one operator and one operand,  $N = 2 \times \text{LOC}$ .

Software metrics are defined using these basic terms. Of interest is another measure for the size of the program, called the volume:

$$V = N \times \log_2 n.$$



The unit of measurement of volume is the common unit for size--"bits." It is the actual size of a program in a computer if a uniform binary encoding for the vocabulary is used. Volume may also be interpreted as the number of mental comparisons needed to write a program of length  $N$ , assuming a binary search method is used to select a member of the vocabulary of size  $n$ . Since an algorithm may be implemented by many different but equivalent programs, a program that is minimal in size is said to have the potential volume  $V^*$ . Any given program with volume  $V$  is considered to be implemented at the program level  $L$ , which is defined by

$$L = V^*/V.$$

The value of  $L$  ranges between zero and one, with  $L = 1$  representing a program written at the highest possible level (i.e., with minimum size). The inverse of the program level is termed the difficulty. That is,

$$D = 1/L.$$

As the volume of an implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as the redundant usage of operands, or the failure to use higher level control constructs will tend to increase the volume as well as the difficulty.

The effort required to implement a computer program increases as the size of the program increases. It also takes more effort to implement a program at a lower level (higher difficulty) when compared with another equivalent program at a higher level (lower difficulty). Thus the effort in software science is defined as

$$E = V/L = D \times V.$$

The unit of measurement of  $E$  is "elementary mental discriminations."

A sound theory should have not only an intuitive set of definitions, but should also contain an intuitive model for which a useful set of hypotheses may be derived and validated. The model, although never explicitly

stated by Halstead, is that most programs are produced by concentrating programmers through a process of mental manipulation of the unique operators and operands. The basic assumption that leads to the hypotheses presented in the following subsections is an implicit limit on the mental capacity of a programmer.

### 3.1 List of Halstead's Metrics

A compendium of Halstead's metrics is presented in Table 3-1. For further explanation of the definitions and terms, the reader is referred to Elements of Software Science by Halstead.

#### Length Equation

$$a. N' = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

#### Volume Metric

$$b. V = N' \log_2 n$$

#### Potential Volume

$$c. V^* = (2 + n_1^*) \log_2 (2 + n_1^*)$$

#### Boundary Volume

$$d. V^{**} = (2 + n_1^* \log_2 n_1^*) \log_2 (2 + n_1^*)$$

#### Relations between Operators and Operands

$$e1. A = (V^{**} - V^*)/V^*$$

$$e2. B = n_1^* - 2(V^{**} - V^*)/V$$

$$e3. n_1 = A n_1 + B$$

$$e4. A = ((n_1^*)/(n_1^* + 2)) \log_2 (n_1^*/2)$$

$$B = n_1^* - 2A$$

#### Program Level

$$f1. L = V^*/V$$

$$f2. L^* = ((n_1^*/n_1)) (n_1/N_2)$$

$$f3. L^* = n_1^*/n_1$$

$$f4. L^* = n_1/N_1$$

#### Intelligence Count

$$g1. I = L^* \times V$$

$$g2. I = ((2/n_1) (n_1/N_1)) (N_1 + N_2) \log_2 (N_1 + n_1)$$

#### Programming Effort

$$h1. E = V/L$$

#### Time Equation

$$i1. T^* = (n_1/N_1) (n_1 \log_2 n_1 + n_2 \log_2 n_2) \log_2 n / 2 n_1 S$$

#### Language Level

$$j1. \lambda = L V^*$$

$$j2. \lambda = L (L \times V) = L \text{ squared times } V$$

#### Time Equation

$$T = EIS$$

#### Error Equation

$$I = (V^*) \text{ squared times } \lambda \text{ squared}$$

$$H = E/E(0)$$

where  $E(0)$  = mean no. of elementary discriminations

between partial errors in programming and  $H$  = no. delivered errors

Table 3-1. Halstead's Metrics

### 3.2 Description of Operators and Operands

Halstead has defined relationships between the number of operator and operand counts in an algorithm to determine several quality factors of the related software. The units from which Halstead's metrics are defined (operators ( $n_1$ ) and operands ( $n_2$ )) can be determined by an automated basis. The following example illustrates what is meant by Halstead's operator and operand count.

#### Interchange Sort Program

```
SUBROUTINE SORT (X,N)
DIMENSION X(N)
IF (N.LT.2) RETURN
DO 20 I=2,N
  DO 10 J=I,I
    IF (X(I).GE.X(J))GO TO 10
    SAVE=X(I)
    X(I)=X(J)
    X(J)=SAVE
10  CONTINUE
20  CONTINUE
RETURN
END
```

### Operator Count and Frequency of Use

Operator	Frequency
1 End of statement	7
2 Array subscript	6
3 =	5
4 IF( )	2
5 DO	2
6 .	2
7 End of program	1
8 .LT.	1
9 .GE.	1
$n_1 = 10$ GO TO 10	1
	$28 = N_1$

### Operand Count and Frequency of Use

Operand	Frequency
1 X	6
2 I	5
3 J	4
4 N	2
5 2	2
6 SAVE	2
$n_2 = 7$ 1	1
	$22 = N_2$

To determine the programming effort, E, the following steps would be exercised:

- (1) Compute the Volume Metric, V

$$V = N \times \log_2 n$$

- (2) Determine an estimate of the program level, L

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

- (3) Compute Programming Effort, E

$$E = V/L$$

### 3.3 Program Difficulty Used to Support Testability

Of the relationships defined, the predicted difficulty, D, is the one best correlated to actual data and most useful for test. The formula describing difficulty may be expressed as:

$$D = \frac{n_1}{2} \times \frac{N_2}{n_2} = 1/L$$

Difficulty then, is the product of two ratios,  $n_1/2$  which increases with the number of operators used, and  $N_2/n_2$  which is the average operand useage. The more times an operand is used, the greater the likelihood for error and the larger the resulting value of D. Thus as D increases, the error proneness of the program increases.

An average number of operators can be determined for the system and an operand useage determined. By using these values, a lower bound of acceptability for D is determined. If the standard deviation of  $n_1$  is added to the average number of operators and D recomputed, the upper limit of D is fixed.

A study at IBM using 30 program modules quantified D for PL/S programs. In these programs, the average value of n1 was 46, the ratio N2/n2 was approximately 5 and the standard deviation of n1 was 18. These values yield upper and lower limits of D as follows:

$$D_1 = \frac{46}{2} \times 5 = 115 \quad (\text{lower bound})$$

$$D_2 = \frac{46 + 18}{2} \times 5 = 160 \quad (\text{upper bound})$$

The boundary conditions above show the points at which the program becomes suspect due to "error-proneness". For instance, a module with a value D greater than 160 should have a team review of concepts and implementation. A module with a value of D between 115 and 160 would suggest that the programmer needs to review his implementation. For D less than 115, the error-proneness of the program is acceptable.

This static test, when applied to a program prior to test should prove to be a valuable tool in predicting maintainability of software. The lower the value of D, the more maintainable a program will be.

### 3.4 Overall Evaluation of Software Science

Halstead's software science is appealing, much of the experimental evidence is convincing, and the psychological foundations are reasonably sound. It is reported to be the most complete attempt at the quantitative evaluation of software. However, it should be noted, that it is not complete since it ignores specific issues, such as choice of mnemonic variable names, comments, control flow complexity, and data structures. Furthermore, general issues, such as portability, flexibility, and efficiency are not addressed.

Halstead's quantitative metrics have had only limited application in actual practice. However, Software Science shows promise as a quantitative tool for software reliability, software development and maintenance effort estimation. Halstead's work should be included in testing as a formal measure of complexity and possibly modularity.

#### 4.0 MCCABE'S SOFTWARE COMPLEXITY METRICS

Thomas McCabe proposed three metrics in IEEE Transactions on Software Engineering in December 1976[2]: cyclomatic, essential, and actual complexity. All three are based on a graphical representation of the program being tested. The first two metrics are calculated from the program graph, while the third metric is calculated at run time.

##### 4.1 Cyclomatic Complexity

McCabe defines cyclomatic complexity by finding the graph theoretical "basis set." In graph theory, there are sets of linearly independent program paths through any program graph. A maximal set of these linearly independent paths, called a "basis set," can always be found. Intuitively, since the program graph and any path through the graph can be constructed from the basis set, the size of this basis set should be related to the program complexity. From graph theory, the cyclomatic number of the graph,  $V(G)$ , is given by

$$V(G) = e - n + p$$

for a graph  $G$  with number of nodes  $n$ , edges  $e$ , and connected components  $p$ . For further explanation of the definition and terms, the reader is referred to the previous cited reference[2]. The number of linearly independent program paths through a program graph is  $V(G) + p$ , a number McCabe calls the cyclomatic complexity of the program. Cyclomatic complexity,  $CV(G)$ ,

where

$$CV(G) = e - n + 2p,$$

can then be calculated from the program graph. In the graph of Figure 4-1,

$$e = 18, n = 14, \text{ and } p = 1.$$

Thus

$$V(G) = 5 \text{ and } CV(G) = 6.$$

A proper subgraph of a graph  $G$  is a collection of nodes and edges such that, if an edge is included in the subgraph, then both nodes it connects in the complete graph  $G$  must also be in the subgraph. Any flow graph can be reduced by combining sequential single-entry, single-exit nodes into a single node. Structured constructs appear in a program graph as proper subgraphs with only one single-entry node whose entering edges are not in the subgraph, and with only one single-exit node, whose exiting edges are also not included in the subgraph. For all other nodes, all connecting edges are included in the subgraph. This single-entry, single-exit subgraph can then be reduced to a single node.

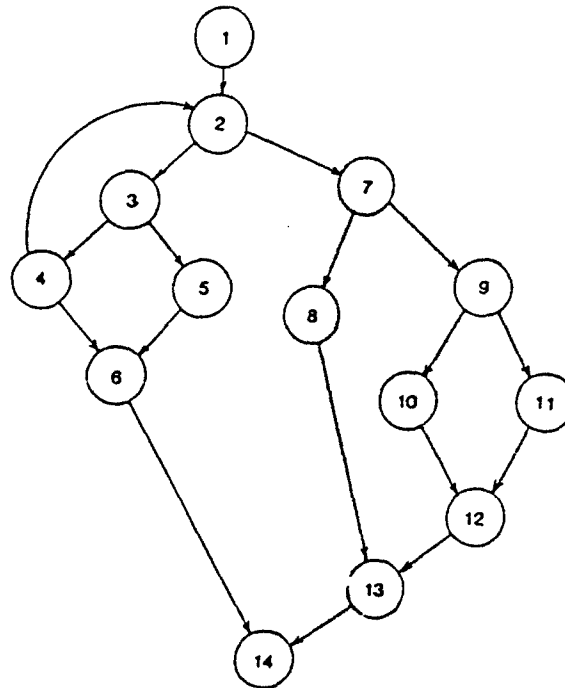


Figure 4-1. Control Flow Graph

#### 4.2 Essential Complexity

Essential complexity is a measure of the "unstructuredness" of a program. The degree of essential complexity depends on the number of single-entry, single-exit proper subgraphs containing two or more nodes.



There are many ways in which to form these subgraphs. For a straight-line graph (no loops and no branches), it is possible to collect the nodes and edges to form from 1 to  $n/2$  ( $n$  = number of nodes) single-entry, single-exit subgraphs. An algorithm can be developed to find the minimum number,  $m$ , of such subgraphs in a graph.

The essential complexity  $EV(G)$  is defined as

$$EV(G) = CV(G) - m$$

where  $m$  is the minimum number of subgraphs in a graph.

Figure 4-2 is an example of a program graph with single-entry, single-exit proper subgraphs identified from such an algorithm. The nodes in the two proper subgraphs are (7, 8, 9, 10, 11, 12, 13), and (9, 10, 11, 12). Note that the second subgraph is entirely contained within the first subgraph. Thus the essential complexity is

$$EV(G) = CV(G) - 2 = 4$$

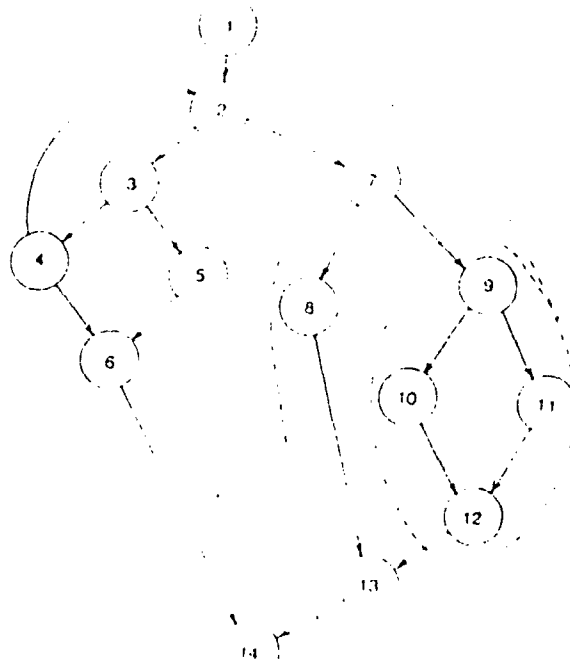


Figure 4-2. Reducible Flow Graph

The program graph for a program built with structured constructs will generally be decomposable into subgraphs that contain single entry and a single exit. The minimum number of such proper subgraphs is  $CV(G) - 1$ . Hence, the essential complexity of a structured program is 1.

#### 4.3 Actual Complexity

Actual complexity, AV, is the number of independent paths actually executed by a program running on a test data set. AV is always less than or equal to the cyclomatic complexity and is similar to a path coverage metric. A testing strategy would be to attempt to drive AV closer to  $CV(G)$  by finding test data which cover more paths or by eliminating decision nodes and reducing portions of the program to in-line code.

#### 5.0 KNOTS COMPLEXITY MEASURE

The "knots" complexity measure, suggested by Woodward, Hennel, and Hedley[3], is based on control flow arcs drawn on the actual sequential source program. Essentially, what is done is to draw arcs from every explicit control flow operator to the destination of that transfer of control. A "knot" is defined to occur when one is forced to draw two such directional arcs that cross each other at some point. It should be clear that the unrestricted use of GO TO statements allowing the production of the much feared "spaghetti" code will have a high number of knots.

Knots complexity can be stated more mathematically and without ambiguity as follows. Let a jump from line a to line b be represented by the ordered pair of integers (a,b). Define  $\min(a,b)$  as the line (a or b) that appears prior to the other. Similarly, the  $\max(a,b)$  is the line (a,b) that appears after the other. As an example, if the  $\max(a,b) = a$ , the jump (a,b) is a backward branch. Then the jump (p,q) gives rise to a "knot" or crossing point with respect to jump (a,b) if either

1)  $\min(a,b) < \min(p,q) < \max(a,b)$   
and  $\max(p,q) > \max(a,b)$

or

2)  $\min(a,b) < \max(p,q) < \max(a,b)$   
and  $\min(p,q) < \min(a,b)$

A measure of software complexity can be obtained by counting the number of "knots" in a program.

To further illustrate the concept of knots, an example of a FORTRAN module with no backward branches and containing four knots is presented in Figure 5-1.

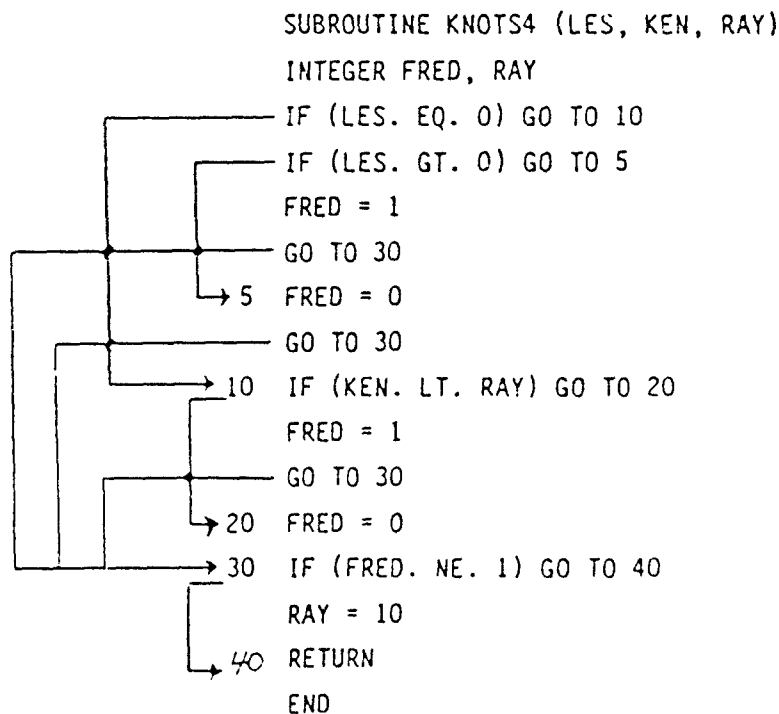


Figure 5-1. Module with Four Knots

## 5.1 Knots Complexity Using Flow Graphs

The definition of knots is dependent on knowledge of program control flow jumps in terms of line numbers. However, it is possible to obtain upper and lower bounds on the number of knots in a program from a directed graph representation provided we also know the ordering of the nodes to make the transition from a two-dimensional graph to a one-dimensional program. (It should be noted that McCabe's Complexity is not dependent on the "location" of the nodes and paths within the code of a program.) The inability to extract the precise number of knots is to be expected, because information concerning the physical source text is discarded in constructing a directed graph of a program.

In order to obtain the lower bound we just apply the definition as it stands, where now the ordered pairs on integers  $(a,b)$  and  $(p,q)$  represent the edges of the directed graph. The strict inequalities in the definition ensure that no knots arise which involve transfer of control to the next node, i.e., edges  $(a,a+1)$ . This is an example of the transfer of control via natural succession. Thus, the number obtained in this fashion is a definite lower bound.

Since the nodes in a directed graph usually represent basic blocks which may involve several source lines with a unique entry point at the head of the block (first line), no branching within the block, and a unique exit point at the tail of the block (last line), there exist situations in which there is insufficient information to resolve the existence or nonexistence of a knot. For example the number of knots in a construction such as in Figure 5-2 depends on whether node B corresponds to several lines (1 knot) or just one line (0 knots). By including all such doubtful cases in our count we can obtain an upper bound to the number of knots.

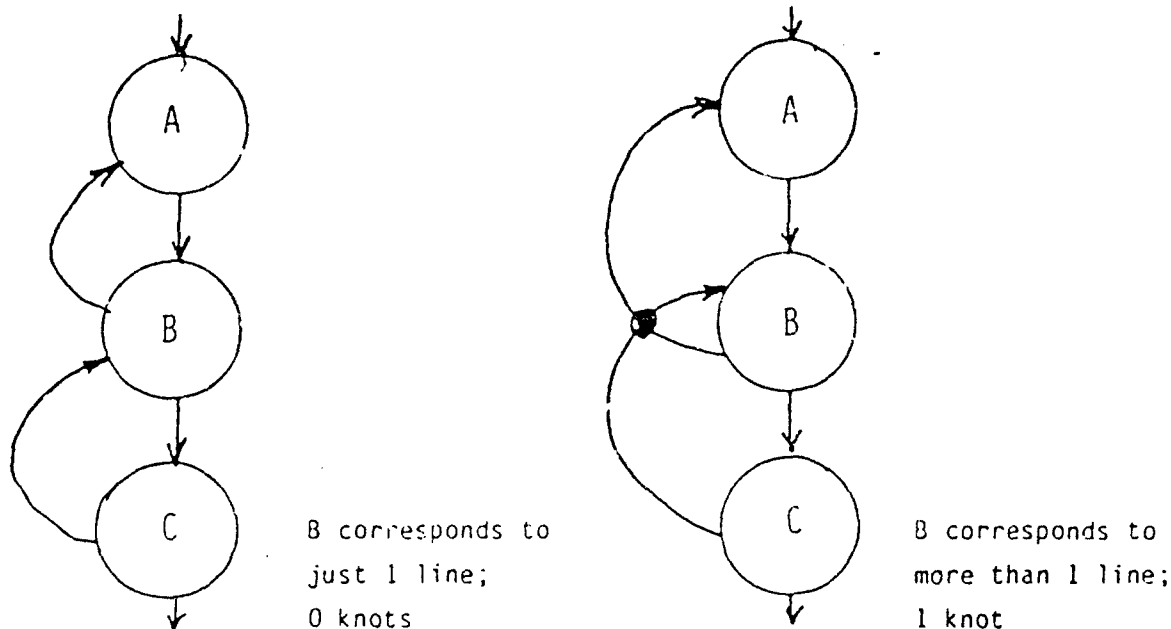


Figure 5-2. Knots Determined from Directed Graph

Figure 5-2 shows the difficulty in determining the number of knots in a program from its directed graph representation. The lower and upper bounds for knots will be shown separated by a colon. In the example above, the representation would be 0:1.

It is to be noted that calculating a lower bound and an upper bound in this way provides a complexity interval in a manner resembling Myers' extension of McCabe's cyclomatic complexity (Section 9).

## 5.2 Knots and Cyclomatic Complexity

One of the advantages the knot count has over the cyclomatic complexity  $V(G)$  is that the number of knots in a program is dependent on the ordering of the statements in the program. Since a directed graph, like a

flow chart, is two dimensional, linearization of it must take place in the actual construction of a program. There will be many ways of ordering the nodes of a directed graph to produce equivalent programs and some will be more complex than others. This will not be reflected in  $V(G)$  since McCabe's measure is independent of the ordering. Consider an example of reordering in Figure 5-3 consisting of 5 nodes and 6 edges. The cyclomatic complexity is the same for both graphs.

$$V(G) = 6 - 5 + 2 = 3.$$

In the first version, the knots interval is 3:4. However, the second restructured (less complex) version has a knots interval of 1:1.

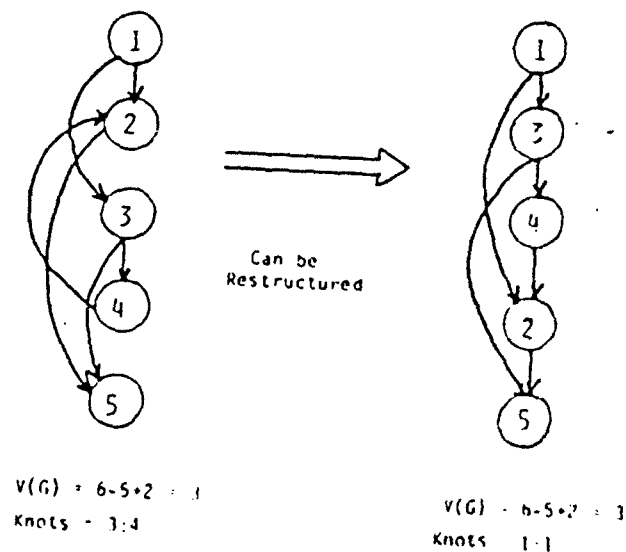


Figure 5-3. An Example Of Reduced Complexity By Restructuring

Other program transformations aimed at code improvement also have the desired property of reducing the knot count. An example is included here in Figures 5-4 and 5-5. It demonstrates once again an advantage of considering knots rather than the cyclomatic complexity alone, since  $v(g) = 3$  for both the original (Figure 5-3) and the undoubtedly less complex version (Figure 5-4) which has no "phantom" paths.

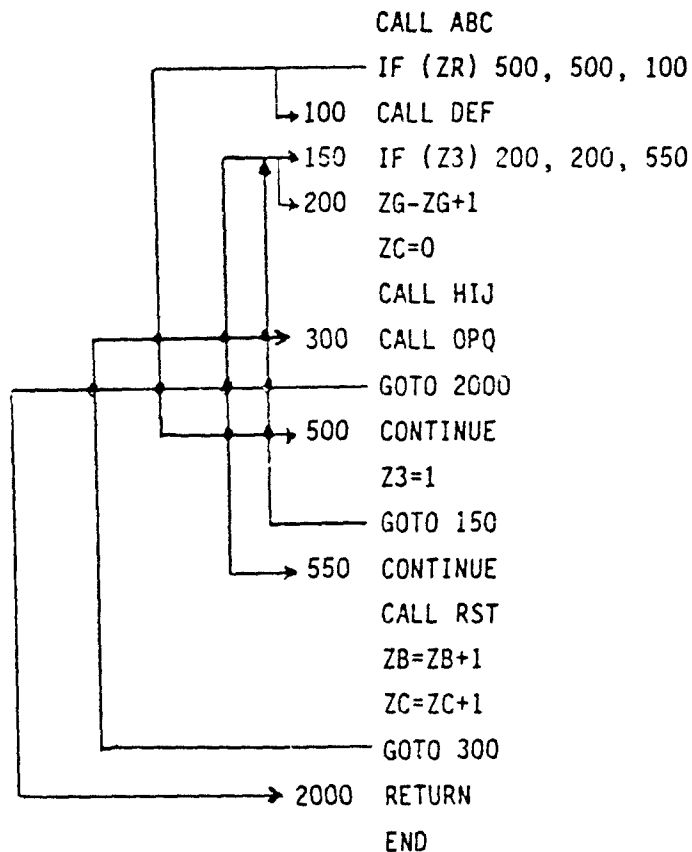


Figure 5-4. Nine Knot Example

Figure 5-4 shows version of code having 9 knots. It has two branch creating statements and so  $V(G) = 2+1 = 3$ .

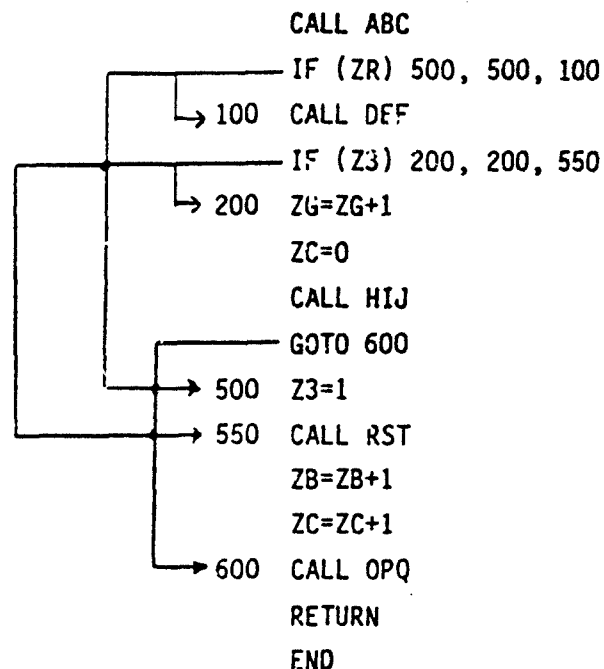


Figure 5-5. Three Knot Example

Figure 5-5 shows the rewritten version of code having no "phantom" paths and 3 knots. It still has two branch creating statements and so  $V(G) = 2+1 = 3$ .

### 5.3 Knots Complexity and Structured Programming

Knots complexity may be used to evaluate structured programming. Since the number of knots is determined from the program text, the complexity of the usual constructs in structured programming will depend upon the language used and the way the constructs are implemented. Consider for example the implementation in Fortran of the simple choice clause:

if bool then . . . A . . . else . . . B . . .



The recommended way of writing this in Fortran is given in Figure 5-6a and has 1 knot. Another way of implementing this construct which does not involve negation of the Boolean expression, but results in more complicated control flow, is given in Figure 5-6b and has 2 knots.

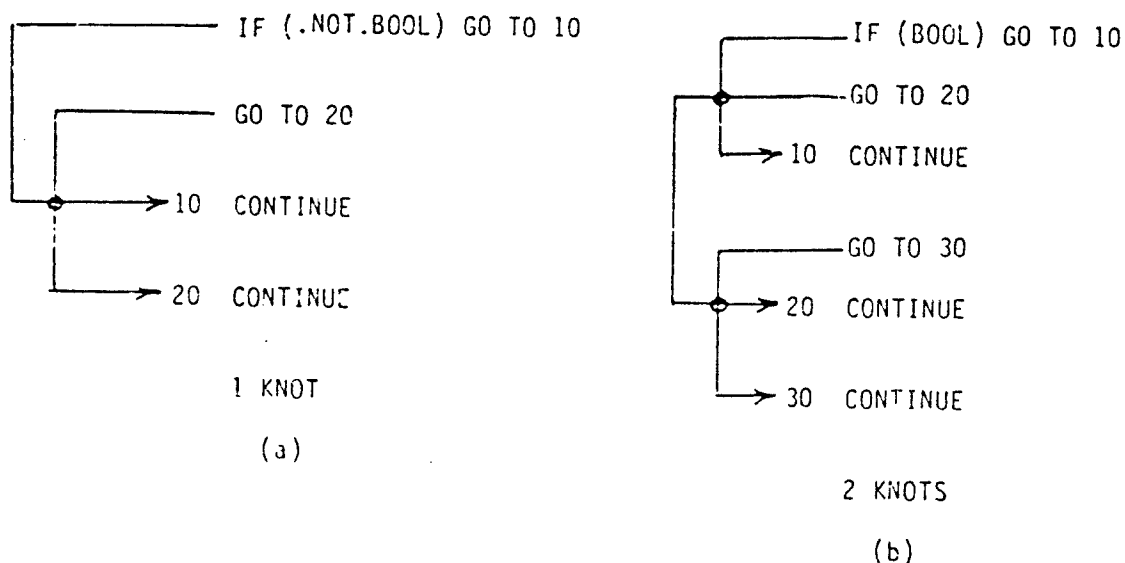


Figure 5-6. Implementation of IF-THEN-ELSE in Fortran

The use of the Fortran arithmetic IF or computed GOTO to provide a 3-way choice results in 3 knots (Figure 5-7). In general, an  $n$ -way case simulated using the computed GOTO will have a knot count of:

$$k = \sum_{i=1}^{n-1} i = \frac{n \times (n - 1)}{2}$$

The while and repeat until looping constructs can be implemented with zero knots.

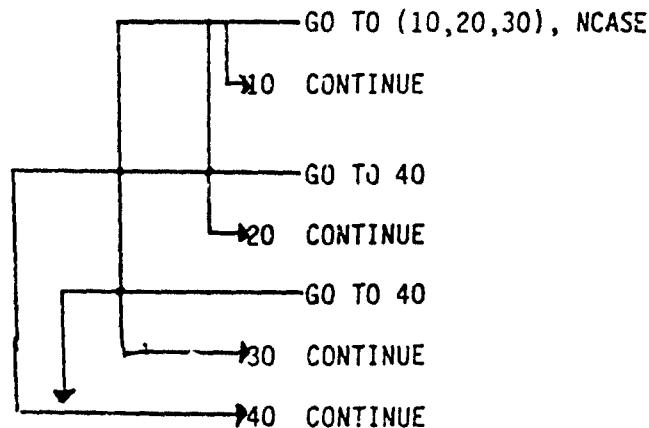


Figure 5-7. Example of 3-Way Case Construct

The contribution to the program complexity arising from the use of structured programming can be removed to provide a measure which reflects the lack of structure in a program. The reduction of the directed graph can be performed by replacing the primitives of structured programming by single nodes. If this process is continued until no further reduction of the graph is possible and the knots interval of the remaining graph  $G'$  is determined, this will provide a measure of program "unstructuredness." A structured program will be reducible to a single node with zero knots. This leads to a definition of the remaining knots as the essential knots of the program and it can be stated that structured program will have zero essential knots. This is analogous to McCabe's calculation of the cyclomatic complexity of the reduced graph  $V(G')$  which equals his essential complexity  $EV(G)$  provided each proper subgraph with unique entry and unique exit is one of the structured programming primitives. Note that, from Section 4,

$$EV(G) = V(G') = 1 \text{ for a structured program.}$$

## 6.0 GILB'S SOFTWARE METRICS

In his book "Software Metrics", Gilb[4] presents a set of basic software metrics, making no claim as to their completeness. He emphasizes that each application requires its own concepts or tools and that his text is intended to provide basic concepts on which the user can build. Gilb builds a strong and convincing case for precise measurement of software based on the history of the physical sciences. Most of these metrics are simply ideas of what might be measured in software evaluation. They are neither definitive nor substantial. They are included here since they present some interesting possibilities, provide a good compendium of definitions for software metrics, and is a pioneering effort in the field.

Gilb discusses the software metrics in 6 basic categories:

- 1) Reliability
- 2) Flexibility
- 3) Structure
- 4) Performance
- 5) Resources
- 6) Diverse

These categories are summarized in the following sections.

### 6.1 Reliability Metrics

Among the metrics Gilb mentions is program reliability metrics, which he defines as the probability that a given program operates for a certain period of time without a logical error. The pragmatic measure for program reliability is one minus the ratio of inputs causing execution failures and the total number of inputs.

Maintainability (the probability that a failed system will be restored to operable condition within a specified time) is described as a function of system design (diagnostic aids, documentation, recovery procedures), personnel, and support facilities such as diagnostic test tools.

Repairability is distinguished from maintainability in that all resources (tools, people, parts) are assumed to be immediately available. It depends more on the nature of the object being repaired.

The characteristic of serviceability (the ease or difficulty with which a system can be repaired) is described by Gilb, but it is not considered quantifiable at present. It is related to repairability.

Availability of a system is computed by dividing the time actually available by the time that the system should have been available. Gilb defines intrinsic availability, operational availability, and use availability.

The attack probability is an expression of the frequency with which latent problems occur. Examples of attacks are sabotage, invalid data values, program logic errors, or even a breakdown of a computer's air conditioning.

Gilb illustrates a scale for generating a risk measure for a database in this discussion of the sensitivity characteristic. Related characteristics include: security probability (the probability of rejecting an attack); integrity probability (probability of system survival); and the accessibility (ease of access to a system) or security measure.

The ratio of correct data to all data is given by Gilb as a measure of the Accuracy (freedom from error). As did Boehm, Brown and Lipow[5], Gilb states that accuracy is necessary for reliability. Precision is defined as the degree to which the errors tend to have the same sort of cause. Gilb measures it by the ratio of the number of actual bugs at the source to the number of corresponding root bugs observed in total which are caused by source bugs. For example, if one error (bug) causes 100 error messages during a specific time, the precision equals 0.01.

## 6.2 Flexibility Metrics

Gilb's second category is flexibility metrics which includes: logical complexity, built-in complexity, open-ended flexibility (adaptability), tolerance (of system input variance), generality, portability, and compatability. For the logical complexity, Gilb proposes a measure of 'binary decisions' in the logic. Such a measurement may be made manually or automatically. The number of nonnormal exits from a decision statement gives the absolute logical complexity. Gilb suggests that logical complexity has been found to be of significance in predicting the cost of computer programs.

Built-in flexibility (the ability of a system to handle different logical situations) is the ratio of usable complexity to total complexity. A built-in flexibility of one indicates that all complexity is desired. This measure is applicable in judging the suitability of software packages against alternatives.

Open ended flexibility (or the measure of the ease with which new functions can be added to a system) may be grossly measured by counting the linkages between modules. Whether or not this indicator is meaningful has yet to be demonstrated.

Tolerance (the ability of a system to withstand a degree of variation in input without malfunction or rejection) is the number of permissible variations which a system will handle 'sensibly.' Tolerance may be designed into a system. There is, however, a tradeoff between the cost of tolerance and the cost of failure due to lack of tolerance.

Gilb's metric of generality (degree to which a system is applicable to a different environment) varies between 0.0 and 1 (for 100 percent.).

For portability,  $P(G)$ , (ease with which a system can be moved from one environment to another) Gilb gives the following measure:

$$P(G) = 1 - E(T)/E(R)$$

where:

$E(T)$  = resources needed to move a system to a target environment;

$E(R)$  = resources needed to create the system for the resident environment.

The portability concept is useful in evaluating the effectiveness of a system that is expected to be moved.

Compatability (the measure of portability that can be expected of systems when they are moved from one given environment to another) must be measured by average, maximum, and minimum portability measures because only portability (not compatability) is measurable in practice.

### 6.3 Structure Metrics

Gilb's structure metrics include: redundancy ratio, hierarchy, structural complexity (modularity) and simplicity, and distinctness.

The redundancy characteristic is a relative one which is measured by the ratio of a quantity for the system being measured to the quantity for some reference system. A redundancy of 1.0 indicates minimum redundancy. An example of the redundancy ratio follows. If a code for a series of 1000 objects is 6 digits 000000-999999 then the digit redundancy is 6/3 or 2.0.

Basic measures of a hierarchy are: depth (number of levels); total number of elements or nodes in the hierarchy; and breadth (number of elements at one level). These measures are related to those of structural complexity which is measured by the number of subsystems or modules and structural simplicity which is the ratio of the number of module linkages to the number of modules. The usefulness of these concepts has not been demonstrated with quantitative data. Rather, 'intuitive' reasoning has been used in the literature when describing these concepts.

Gilb defines distinctness as a measure of the failure-point independence of one piece of software with that of another piece performing the same function. The measuring tool is the ratio of the number of bugs in the first module alone to the number of bugs in the first module which simultaneously occur in the second. The following classes of errors are purportedly detectable and correctable using distinct software: design errors leading to system errors, resource problems, numeric errors, order code error, timing errors, data transmission and software errors.

#### 6.4 Performance Metrics

The performance metrics described by Gilb are: efficiency, effectiveness, and transformation.

Efficiency is defined as the ratio of useful work performed to the total energy expended, while effectiveness is a group measure comparing: operational reliability, system readiness and design adequacy. Efficiency and effectiveness are tradeoffs against each other. For example, it might be very effective to do triple verification of data registration by operators but it would be very inefficient.

A transformation measure is the energy or resources needed to convert data from one state to another. The measurement may be in such terms as money, time, personnel, electricity, or logical cycles. The measure is important to measure the effect (in performance) of changes to algorithms, modules, etc.

#### 6.5 Resource Metrics

Gilb's resource metrics are: financial datametrics, time resource datametrics, and space metrics. Financial datametrics include: total system cost, incremental costs, capital investments, operational costs, and return on investment. Time resource datametrics include measures of computer time and personnel time. Space metrics involve the amount of space (bits, words, characters, etc) for storing data.

## 6.6 Diverse Metrics

Finally, Gilb includes diverse metrics of information, data, evolution, and stability. Information is the interpretation of data, and is not directly measurable. Data has many possible forms which should be taken into account when measuring. Bits are useful measures of data when relating to machine cost. Evolution (the designed characteristic of a system development which involves stepwise change) may be measured by such indicators as: number of program instructions changed; percentage of instruction manuals changed; number of new data elements in a database, etc.

Stability (the measure of lack of perceivable change in spite of an occurrence which would normally cause change) is given as a percentage of change in a system due to a change in environment. An example would be the percentage of the original number of payroll programs changed due to a change in tax laws.

## 6.7 Summary of Gilb's Metrics

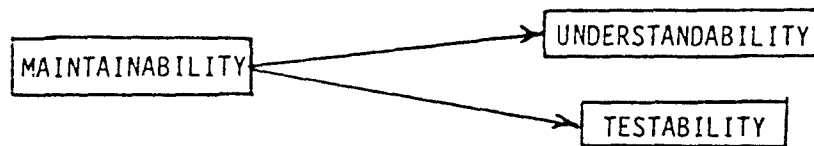
This lengthy enumeration of metrics is more provocative than productive. Gilb's work does open up ideas to fresh possibilities, but the reality of applying many of these ideas is disheartening. Many of the metrics are difficult to obtain, and even if we can compute a value, we have no sense of the range of good values. The lack of independence of the metrics adds to the confusing complexity and makes it difficult for programmers to predict the effect of a program change on a group of metrics.



## 7.0 BOEHM'S QUANTITATIVE EVALUATION OF S/W QUALITY

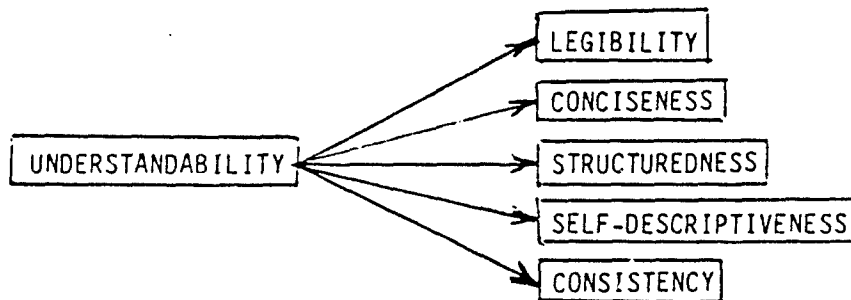
Boehm has developed a set of S/W quality characteristics that are subjectively interrelated to a set of metrics which are quantifiable and thus can provide a measure of S/W quality. These are best expressed in the form of a tree.

The elements of the tree, such as:



are related in the direction of the arrows, i.e. if a program is maintainable, it must be both understandable and testable.

It would follow, then, that a lower level of primitive concepts exist below understandability and testability that are quantifiable (metrics). These are expressed as:



The entire concept thus becomes:

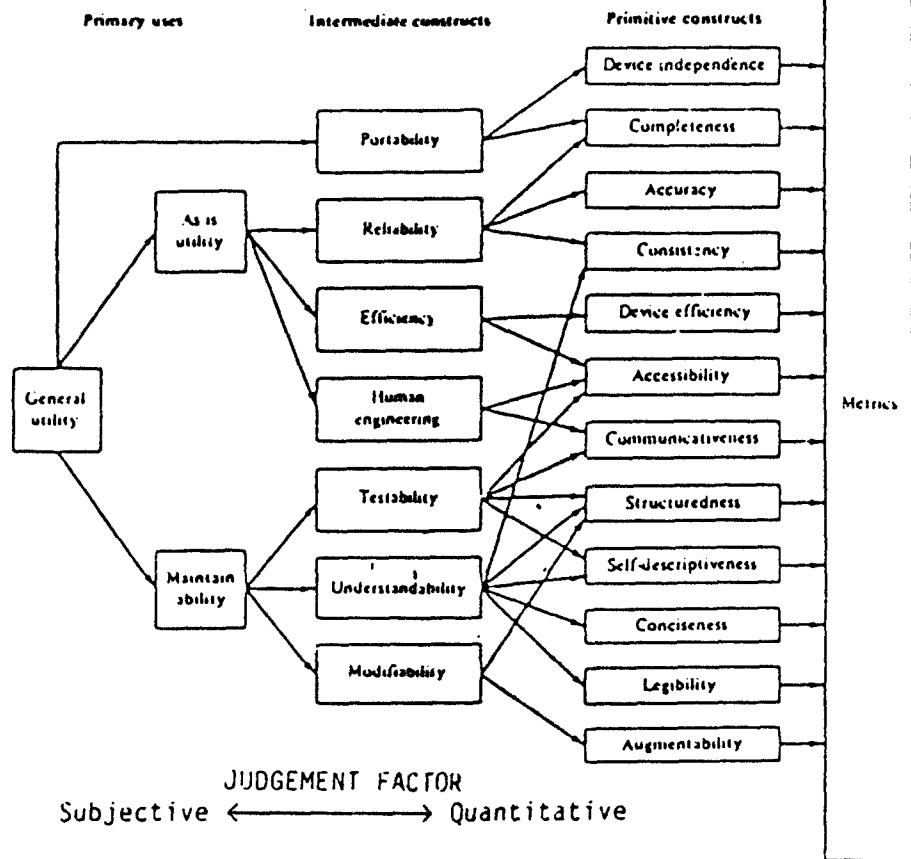


Figure 7-1. Software Quality Characteristics Tree (Ref. 7)

The tree illustrates a logical progression of evaluating a conceptual quality by assigning a quantitative numerical value to measureable metrics.

The degree of correlation of the metrics selected to the element of software quality that we wish to quantify becomes important, since if the correlation is not high, the values attained by use of metrics are of questionable benefit to the evaluator.

Additionally, if the metric cannot be quantified using a cost-effective method, the benefits of using S/W metrics is low. Automating the quantification of metrics ranges from straightforward and trivial to virtually impossible. For example, counting average module length is straightforward, but

judging descriptiveness of material is very difficult. Table 7.1 presents a summary of metrics and some of their characteristics.

Table 7.1. Evaluation of Metrics (Ref. 5)

Primitive Characteristics	Definition of Metrics	Correlation with Quality	Potential Benefit	Quantifiability	Ease of Developing Automated Evaluation	Completeness of Automated Evaluation
Device-Independence DI-1	Are computations independent of computer word size for achievement of required precision or storage scheme?	A	5	AL + EX + TI	E	P
DI-2	Have machine-dependent statements been flagged and commented (e.g., those computations which depend upon computer hardware capability for addressing half words, bytes, selected bit patterns, or those which employ extended source language features)?	A	5	AL	M	P
Self-Containedness SC-1	Does the program contain a facility for initializing core storage prior to use?	A	5	AL	E	P
SC-2	Does the program contain a facility for proper positioning of input/output devices prior to use?	A	5	CC	E	P
Accuracy AR-1	Are the numerical methods used by the program consistent with application requirements?	A	5	TI		
AR-2	Are the accuracies of program constants and tabular values consistent with application requirements?	A	5	AL + TI	E	P
Completeness CP-1	Are all program inputs used within the program or their presence explained by a comment?	U	3	AL	E	C
CP-2	Are there no "dummy" subprograms referenced?	S	2	AL	E	C
Robustness R-1	Does the program have the capability to assign default values to non-specified parameters?	A	5	AL + TI	E	P
R-2	Is input data checked for range errors?	AA	5	AL + TI	E	P
Consistency CS-1	Are all specifications of sets of global variables (i.e., those appearing in two or more subprograms) identical (e.g., labeled COMMON)?	AA	4	AL	E	C
CS-2	Is the type (e.g., real, integer, etc.) of a variable consistent for all uses?	A	5	AL	E	P

1. Correlation with Software Quality.

- A - Very high positive correlation; nearly all programs with a high metric score will possess the associated characteristic.
- AA - High positive correlation; a good majority (say 75-90%) of all programs with a high metric score will possess the associated characteristic.
- U - Usually (say 50-75%) of all programs with a high metric score will possess the associated characteristic.
- S - Some programs with high metric scores will possess the associated characteristic.

2. Potential Benefit of Metrics.

- 5 - Extremely important for metric to have a high score; major potential troubles otherwise.
- 4 - Important for metric to have a high score.
- 3 - Fairly important for metric to have a high score.
- 2 - Some incremental value for metric to have high score.
- 1 - Slight incremental value for metric to have high score; no real loss otherwise.

3. Metric Quantifiability and Feasibility of Automated Evaluation.

AL - Can be done cost-effectively via an automated algorithm.

CC - Can be done cost-effectively via an automated compliance checker if given a checklist (Code Auditor is such a tool).

UI - Requires an untrained inspector

TI - Requires a trained inspector

EI - Requires an expert inspector

EX - Requires program to be executed

Ease of Developing Automated Evaluation

E - Easy to develop automated algorithm or compliance checker.

M - Moderately difficult to develop automated algorithm or compliance checker.

D - Difficult to develop automated algorithm or compliance checker.

Completeness of Automated Evaluation

C - Algorithm or checker provides total evaluation of metric.

P - Algorithm or checker provides partial evaluation of metric.

I - Algorithm or checker provides inconclusive results.

Only a few of the candidate metrics are displayed in Table 7-1, and this does not imply a complete listing of available metrics for measuring the primitives defined by Boehm.

An expansion of Boehm's work was done by McCall, et al[9], and the list of quality factors was expanded from 7 to 11 (Figure 7-2), which is generally accepted as complete today.

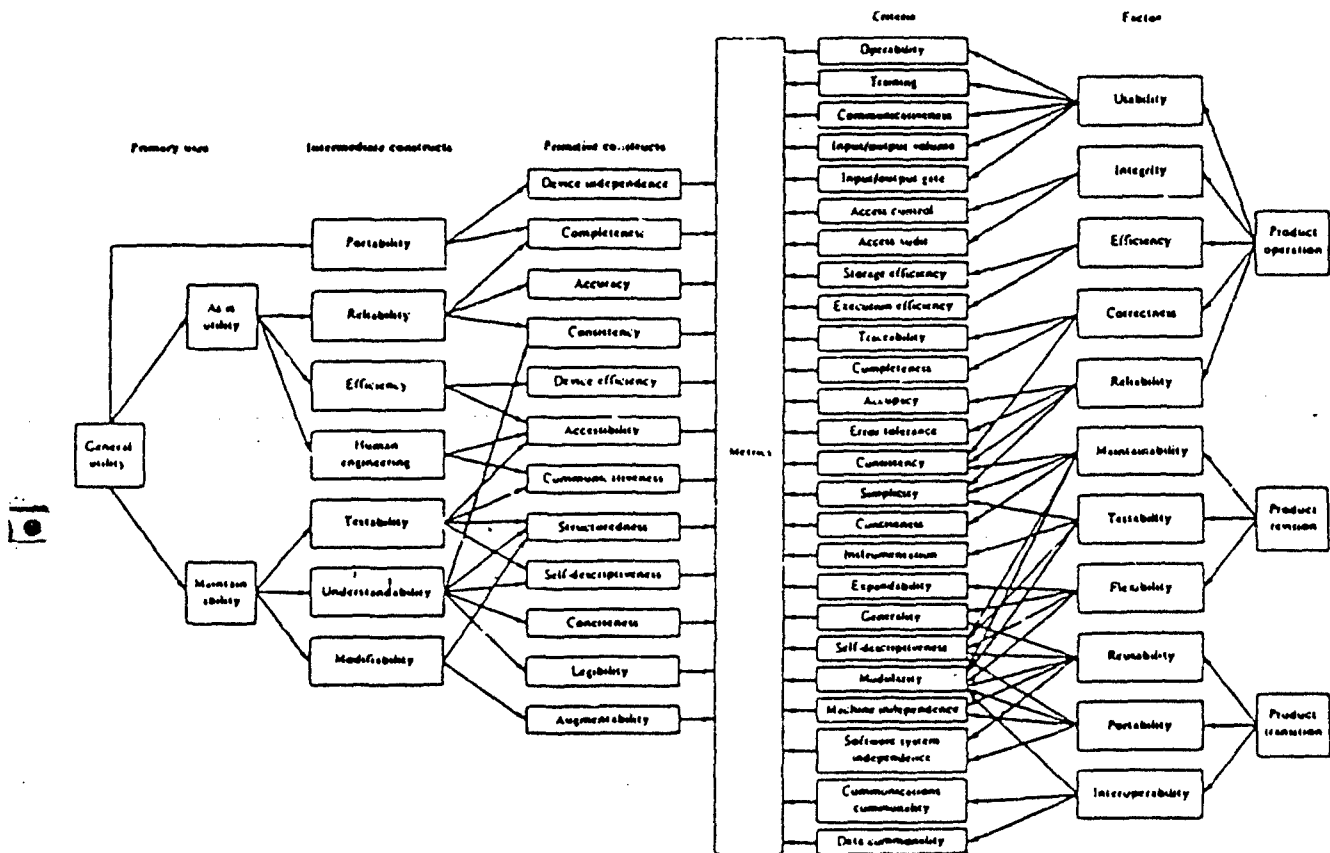


Figure 7-2. Software Quality Comparisons (Ref. 7)

The work by Boehm[5] and McCall[9] address the measurement of subjective qualities of software programs by breaking broad terms down into measurable and testable items. The list of metrics presented by McCall parallels Boehm's work so closely that only a comparison of the results are presented here.

## 8.0 THAYER'S SOFTWARE RELIABILITY

Thayer[6] has taken existing S/W reliability models and correlated the error history of several programs to produce a reliability prediction using established mathematical models. The data collection is during the test phases of the software life cycle. Figure 8-1, from Thayer, illustrates the software life cycle and where problem reporting and error data is collected.

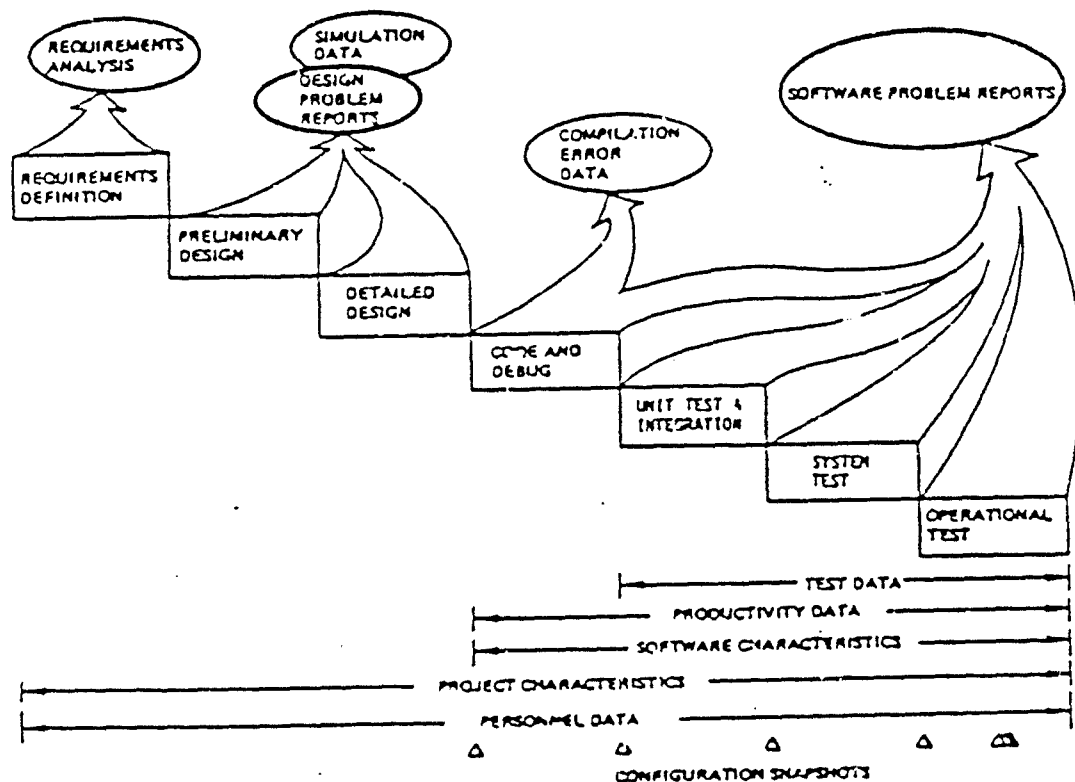


Figure 8-1. Data Availability Throughout the Software Development Cycle (Ref. 6)

The concept of evaluation of S/W error data has evolved to a comprehensive short listing that is easy to use and understand. The evolution to Thayer's list of major error categories is shown in Table 8-1. The first column depicts the number of major error categories and column 2 shows the total of the detailed error categories.

Table 8.1. History of Error Category Lists (Ref. 6)

Major Categories	Detailed Categories	Iteration	Comments
13	0	Study done in support of CCIP-85	<ul style="list-style-type: none"> <li>• Entirely symptomatic</li> </ul>
13	224	Post-CCIP-85 in-house work	<ul style="list-style-type: none"> <li>• Greater emphasis on cause</li> <li>• Failed to recognize code types</li> </ul>
20	164	Interim technical report for Software Reliability Study	<ul style="list-style-type: none"> <li>• Predominately symptomatic</li> <li>• Recognized types of code</li> <li>• Assignment not by problem fixer</li> </ul>
25	435	Early causative work	<ul style="list-style-type: none"> <li>• Generated by speculation</li> <li>• Long with redundancies</li> <li>• Hard to use</li> </ul>
12	79	Final causative list	<ul style="list-style-type: none"> <li>• Comprehensive but short</li> <li>• Easy to use</li> <li>• Problem fixer assigns categories</li> </ul>

Thayer's list of 12 major error categories and the number of detailed categories in each are:

- (1) Computational errors - contains 9 detail categories.
- (2) Logic errors - contains 7 detail categories.
- (3) Data input errors - contains 6 detail categories.
- (4) Data handling errors - contains 10 detail categories.
- (5) Data output errors - contains 8 detail categories.
- (6) Interface errors - contains 7 detail categories.
- (7) Data Definition errors - contains 4 detail categories.



- (8) Data base errors - contains 3 detail categories.
- (9) Operation errors - contains 6 detail categories.
- (10) Other - contains 9 miscellaneous detail categories.
- (11) Documentation errors - contains 5 detail categories.
- (12) Problem report Rejection - contains 5 detail categories.

The problems reported by Thayer, et al, in data collection and analysis are equally valid for forms of data collection other than error history and are well worth noting:

Table 8.2. Data Collection and Analysis Problems (Ref. 6)

- 1. Projects, the software, and the data vary considerably and are not describable in common terminology.
- 2. Data collection can represent cost, schedule, and manpower impediments to software development projects. The impact or cost considerations of data collection, although real, are not fully appreciated.
- 3. Data collection is a lot of work. The tools and techniques for collecting data are not available.
- 4. Certain data items are perishable and must be collected and analyzed when they become available, not after the fact.
- 5. Performers, project management, and even the buyers of software are sensitive about providing data that might be used to adversely evaluate the project by external agencies.
- 6. Some projects produce data that are classified.
- 7. Analysis techniques and questions to ask of the data are not well known.
- 8. There is no guarantee that data will be collected (i.e., no requirement for projects to collect data).
- 9. Data accuracy is a chronic question.
- 10. Analysis is often incomplete or inaccurate if proper communication with project performers is not established.
- 11. Contractor and customer representatives of project management are not aware of the benefits of data analysis and therefore tend not to support it.
- 12. Project structure is generally not tailored to use available data (i.e., the mechanism for analyzing data and folding results back into the project is not provided).
- 13. The fervor of data collection inspires data gathering that is non-supportive of the software development process.
- 14. Some data elements require protection to preserve the privacy of the contributor (e.g., cost data).
- 15. Data collection is commonly thought to be "not necessary" to a properly managed project.
- 16. Project organizational structure and resources vary, making consistent, multi-project data collection questionable.
- 17. Definition of which parameters are needed and meaningful to collect is in its infancy.
- 18. Presently implemented data collection schemes often fail to gather data in sufficient detail, making results of analysis questionable.

The models used by Thayer, et al, for evaluation of error data are the well documented models of Shooman, Jelinski and Moranda, and Schick. The analysis, though interesting and well done is not applicable to pre-test data collection and evaluation. Therefore, the use of Thayer's reliability estimates is very limited.

#### 9.0 MYERS' EXTENSION TO MCCABE'S CYCLOMATIC COMPLEXITY

Myers reported (SIGPLAN 1977) on an anomaly that occurs when using different techniques to determine McCabe's cyclomatic complexity. The problem arises since there are two ways of drawing flow diagrams for compound IF statements. The following example is used to illustrate Myers' concern:

- A) IF (X.EQ.O.) THEN  
ELSE
- B) IF (X.EQ.O..AND.Y.GT.O.) THEN  
ELSE
- C) IF (X.EQ.O.) THEN  
IF (Y.GT.O.) THEN  
ELSE

The hypothesis is that A is less complex than B and that B is less complex than C.

If McCabe's cyclomatic complexity  $V(G)$  is computed by diagramming statements or counting whole predicates as decisions, one gets

$$V(A) = 2 < V(B) = 2 < V(C) = 3$$

Similarly, if one computes  $V(G)$  by counting individual conditions (plus one), then

$$V(A) = 2 < V(B) = 3 < V(C) = 3$$

Neither inequality agrees precisely with the hypothesis, so Myers suggested combining the calculations and displaying a complexity interval. Such that,

$$V(A) = (2,2) < V(B) = (2,3) < V(C) = (3,3)$$

While this is mathematically appealing, it doesn't appear to be essential for determining module complexity.

However, the complexity interval is easy to calculate, provides information regarding programming technique (maintainability), and eliminates continually justifying the computational choice of  $V(G)$ . Therefore, it is recommended that software analysis tools be extended to include this.

It should be noted that since Halstead's technique (Section 3) is based on use of operators and operands, the above anomaly is handled without ambiguity.

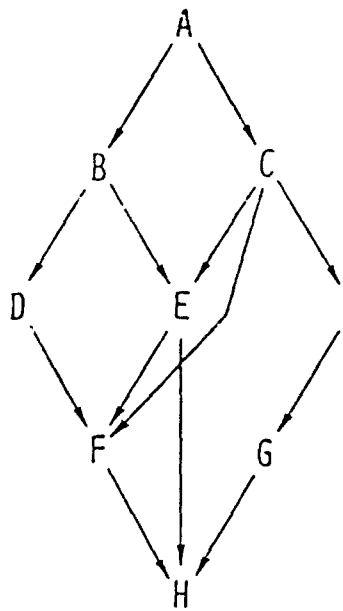
## 10.0 COMPARISON OF HALSTEAD'S, MCCABE'S, AND KNOTS METRICS

The McCabe and knots metrics are based on program control flow. Neither approach penalizes for in-line code complexity. McCabe's complexity is based on graph theory for determination of nodes and paths. These flow graphs are mathematically consistent and are more defensible for test support. Knots supports the determination of "unstructuredness" quite well and, therefore, extends McCabe's cyclomatic complexity. However, McCabe's "Essential Complexity" which determines subgraphs also addresses the issue of structured programming. Another advantage of McCabe's over knots is that McCabe's metrics can be determined without needing the actual module code. Cyclomatic complexity may be determined from directed flow graphs and Program Design Languages (PDLs).

Halstead's software science is based on psychological principles of programmer performance. This approach extends software engineering to the principles of experimental science. It is not based on control flow theory but rather the number of unique operators and operands and the total use of each. Therefore, Halstead does address the issues of in-line code complexity. Software science handles program flow by investigating the code for the inclusion of decision and branching statements.

One approach to studying the worth of metrics for flow complexity issues is to investigate each with respect to accepted methods of flow simplification. Three such methods are 1) Linearization; 2) Node Splitting; and 3) Structuring Multiple Exit Loops.

The main issue of linearization is that the same two dimensional flow graph as shown in Figure 10-1 may be represented by different code (linearization).



SAME GRAPH => DIFFERENT CODE

Figure 10-1. Two Dimensional Flow-Graph

The code produced in different programs may have the same flow graph but may vary in complexity. McCabe's complexity metric will not show this difference but Halstead's will. Knots measure will depend on the particular linearization (Figure 5-3) and also on the implementation language (IF-THEN-ELSE constructs).

Node splitting refers to the process of producing an executionally equivalent flow graph  $G'$  from a flow graph  $G$ , where one node in  $G$  appears more than once in  $G'$ . In many instances, such transformations may be used to produce flow graphs in which each subgraph consists of a single entry arc and a single exit arc (i.e. structured flow graphs). Figure 10-2 is a simple illustration of node splitting where node D is repeated in (b).

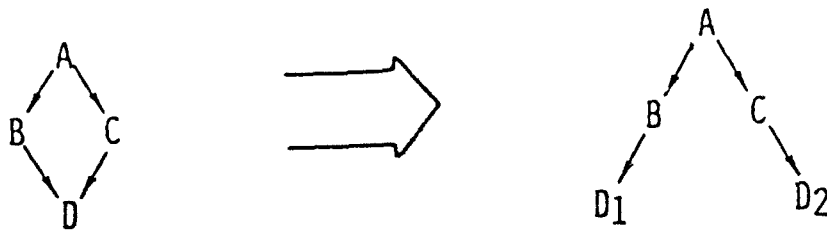


Figure 10-2. Node Splitting

Since node splitting involves code duplication, Halstead's V (and hence E) increase while node splitting is making the control flow less complex. However, this may lead to creation of subroutines so that Halstead's may not be effected greatly. McCabe's cyclomatic complexity is not adversely affected by the duplication of code.

One flow graph construct which cannot be structured using only node splitting transformations is that of a loop with different paths from the loop body to the exit node. There are two general approaches for structuring such flow graphs and both must involve the use of additional program variables to "remember" key program variable values at "critical points" in the execution of the loop body. These values can then be used to determine the paths through the loop and allow for a single exit. A second approach involves the use of Boolean variables.

In the subsequent analyses the restrictive notion of "multiple exit loop" in Figure 10-3 serves as the basis for discussion. This case actually occurs in instances of multiple error exits from a loop. The structured flow graph in Figure 10-3 makes use of the Boolean variable 'check' to predicate the iteration.

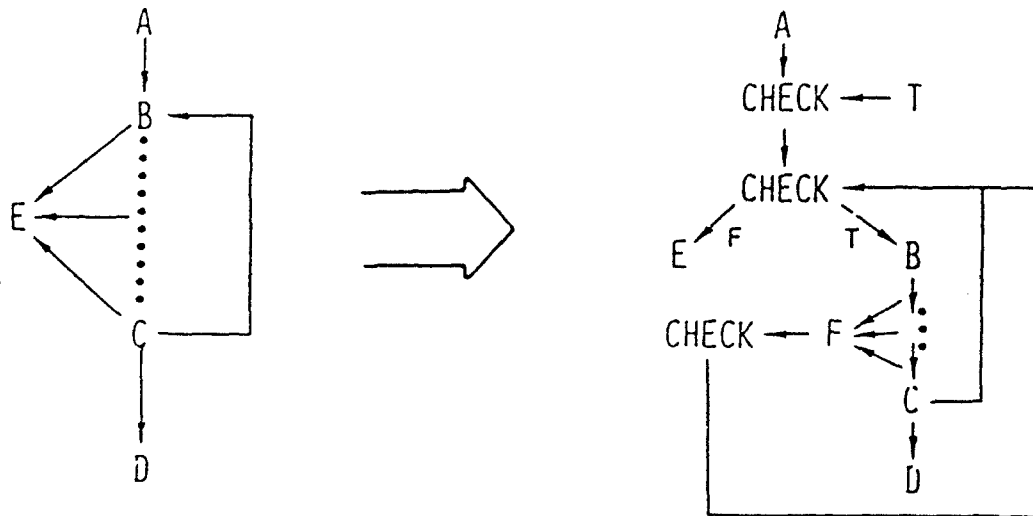


Figure 10-3. Multiple Exits

Here again, the suggested solutions for handling multiple exits involves increasing the amount of code which increases Halstead's metrics.

In the case of structuring multiple exit loops, there is a small increase in control flow (McCabe's) complexity. In particular the increase is fixed at a small level and is not unreasonable. However, the increase in software effort may be quite large.

With respect to any structuring transformation, Knots metric indicates that such transformations are advisable. This follows from the fact that programs which are well structured (in the sense that they are composed using only IF...THEN...ELSE and DO WHILE control flow operators) will have  $K=0$ . Thus, any set of transformations which structure a program must result in a program with a  $K$  value no greater than that of the original program. Given the rather intense level of debate concerning the utility of such transformations, this result should evoke some skepticism of the knots measure.

Supporting evidence has shown that Halstead's metrics support the linearization issue quite well, node splitting somewhat, and multiple exit loops not very well. McCabe's on the other hand does not support linearization well but does support the other two issues. Arguments can be made against knots in all three issues. It may, therefore, be concluded that Halstead and McCabe may complement each other in resolution of the above issues.

A further observation about knots is that programs with arbitrary amounts of structured transfers will have the same complexity as any straight line code. This is not the case for Halstead or McCabe. Knots, however, is the best measure of the occurrence of "spaghetti" code. This is of increasingly less value with the increase and proliferation of languages with structured constructs.



## 11.0 CONCLUSIONS AND RECOMMENDATIONS

Quantitative measures of software quality are still in their infancy. Halstead's software science and McCabe's cyclomatic complexity, currently the best developed software metrics, show promise as predictors for effort estimation, testing, reliability, and maintenance. Woodward's "knots" control complexity best evaluates structured programming constructs and augments McCabe's cyclomatic complexity.

Recent literature has pointed out weaknesses and limitations of each of these metrics in specific situations without offering viable alternatives. The lack of vast empirical data to evaluate/substantiate proposed metrics has hindered the development and acceptance in software engineering. With time, reliable and useful standard measuring concepts will emerge.

The metrics of Halstead and McCabe should be included for software evaluation. McCabe's cyclomatic complexity has been used with some success. This technique should be expanded to also determine "essential" complexity to evaluate structured programming constructs. If this is done, there will be minimal additional value obtained by Woodward's "knots" metric and, therefore, "knots" need not be included. Halstead's and McCabe's metrics, especially utilized in concert, can help to assess the testability and maintainability of software to be implemented. The information required to determine these metrics will undoubtedly be necessary for or similar to information required for other metrics that may be proposed in the future.

In summary, the metrics presented here to assess software quality should be included in a software assessment tool. They are only partially validated and, therefore, must be used with some care. They are intended to quantify information that is extremely subjective. Software type, programming language, timing constraints, and coding efficiency should always be considered when assessing software. However, these metrics, especially if used together, will help determine modules that are overly complex, difficult to test/maintain, potentially error prone, in need of rework, and possibly catastrophic to the system under test.

## 12.0

## REFERENCES

- (1) Maurice H. Halstead, Elements of Software Science, Elsevier North-Holland Inc., New York, 1977
- (2) Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering (SE-2(4), Dec. 1976)
- (3) Martin R. Woodward, Michael A. Hannell, and David Hedley, "A Measure of Control Flow Complexity in Program Text", IEEE Transactions on Software Engineering SE-5(1):45-50, Jan. 1979
- (4) Tom Gilb, Software Metrics, Winthrop, 1977
- (5) B. W. Boehm, J. R. Brown, M. Lipow, "Quantitative Evaluation of Software Quality" Proceedings, 2nd International Conference on Software Engineering, 1976, IEEE
- (6) Thomas A. Thayer, Myron Lipow, and Eldred C. Nelson, Software Reliability: A Study of Large Project Reality, Elsevier North-Holland, New York, 1978
- (7) Bill Curtis "Experimental Evaluation of Software Characteristics" Software Metrics: An Analysis and Evaluation, Alan Perlis, Frederick Sayward, and Mary Shaw (Editors), MIT Press, 1981
- (8) Glenford J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity", ACM SIGPLAN Notices, Oct. 1977
- (9) James A. McCall, "An Introduction to Software Quality Metrics", Software Quality Management, John D. Cooper and Matthew J. Fisher (Editors), Petrocelli Books, Inc., 1979

## APPENDIX I

### SOFTWARE MODULE COUPLING AND STRENGTH

## TABLE OF CONTENTS

	PAGE
1.0 INTRODUCTION .....	1
2.0 SOFTWARE STRUCTURE .....	2
2.1 Structural Definitions .....	4
3.0 MODULARITY .....	6
3.1 Abstraction .....	7
3.2 Information Hiding .....	8
3.3 Module Types .....	9
4.0 MODULE INDEPENDENCE .....	11
4.1 Coupling .....	11
4.1.1 Data Coupling .....	12
4.1.2 Stamp Coupling .....	13
4.1.3 Control Coupling .....	13
4.1.4 Common Coupling .....	14
4.1.5 Content Coupling .....	16
4.1.6 Determining Coupling Type .....	16
4.2 Cohesion .....	17
4.2.1 Functional Cohesion .....	18
4.2.2 Sequential Cohesion .....	18
4.2.3 Communicational Cohesion .....	18
4.2.4 Procedural Cohesion .....	19
4.2.5 Temporal Cohesion .....	19
4.2.6 Logical Cohesion .....	19
4.2.7 Coincidental Cohesion .....	20
4.2.8 Determining Module Cohesion .....	20
4.2.9 Summary of Cohesion .....	20
5.0 REFERENCES .....	22

## 1.0 INTRODUCTION

This report discusses the strength (cohesion) and interconnectivity (coupling) of modules and provides recommendations for assessment of software as a total system.

Unfortunately, there are no currently existing automated techniques to determine module cohesion or coupling. However, this report will discuss the various aspects and terminology applied to modularity, strength, and coupling. It is recommended that this terminology be adopted for use in software assessment. A technique to determine the relative strengths and coupling of modules is presented. Furthermore, system level metrics are summarized for potential application in future test efforts.

## 2.0. SOFTWARE STRUCTURE

Software structure is a hierarchical representation which indicates the relationship between elements (called modules) in a software solution to a problem implicitly defined by requirements analysis. The evolution of software structure begins with the problem definition. Parts of the problem are solved by one or more software elements (modules). This process is symbolically shown in Figure 1 where modules are created to solve different parts of the problem.

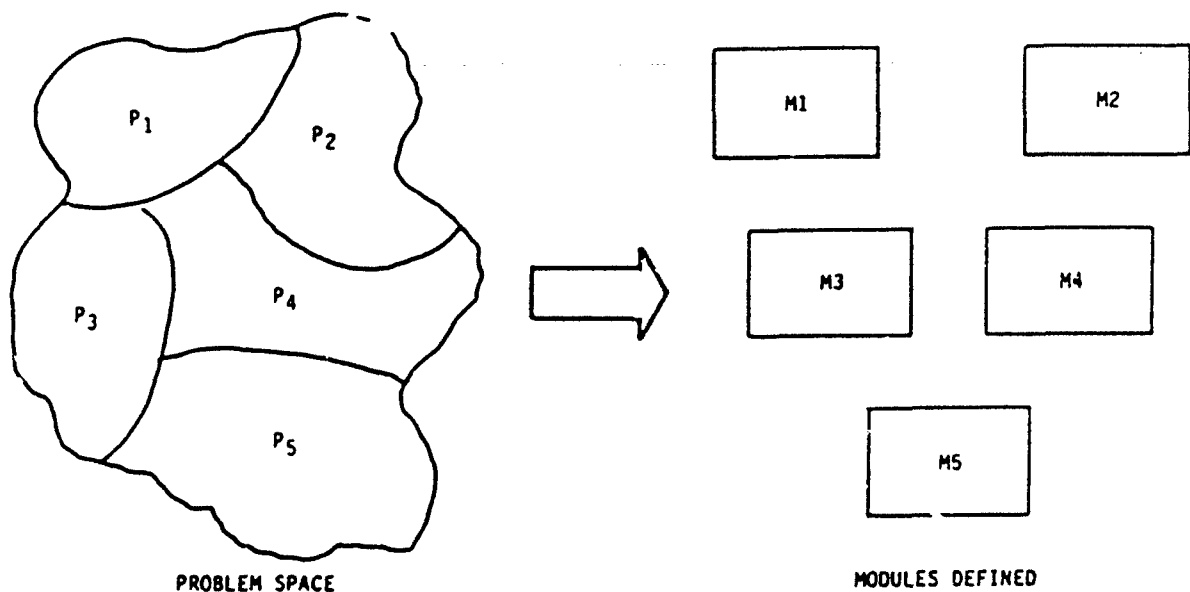


Figure 1. Evolution of Software Structure

Software structure represents a program architecture that implies a hierarchy of control. However, the procedural aspects of software are not represented by structure. For instance, structure does not represent the sequence of processes, the repetition of operations, or the occurrence and order of decisions.

The structure of a software system may be created by utilizing and interfacing modules in different ways. All modules could communicate directly with a single "controlling" module. This design would require all data elements to be defined by the controlling module and "ping-pong" in and out as required. Furthermore, the controlling module will become extremely large and complex for all but simple problems. Software systems should be factored to consolidate flow of control and decision making modules.

By distributing control in a top-down fashion, design and implementation are simplified, testability is enhanced, and maintenance can be approached in a more efficient manner.

It can be seen from Figure 2 that a problem may be solved by many different software structures. Because each is based on different philosophies, each design method will result in a different structure for the same set of software requirements. Unfortunately, there is no magic formula to determine "which is best". However, there are characteristics of a structure that can be examined and should be considered when assessing software. These design methodologies are discussed below and should be considered during software assessment.

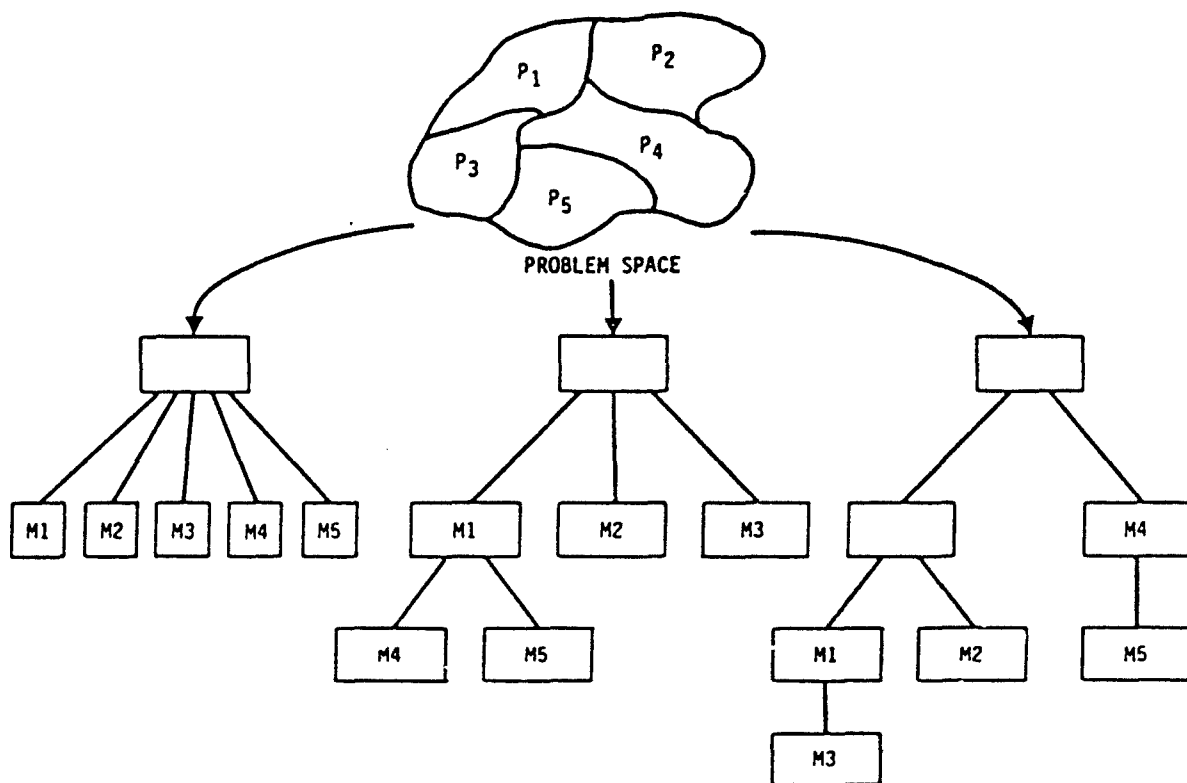


Figure 2. Various Software Designs

## 2.1 Structural Definitions

In order to facilitate discussions on software structure and recommended metrics, a few simple measures and terms are defined. Each of the boxes contained in a structure diagram represents one module -- a separately addressable element of a program (e.g., subroutine, function, procedure). Figure 3 illustrates various measures of structure. Depth refers to the number of layers of control (vertically) and width refers to the overall span of control (horizontally). Fan-out is a measure of the number of modules that are directly controlled by another module. While fan-in indicates how many modules directly control a given module.



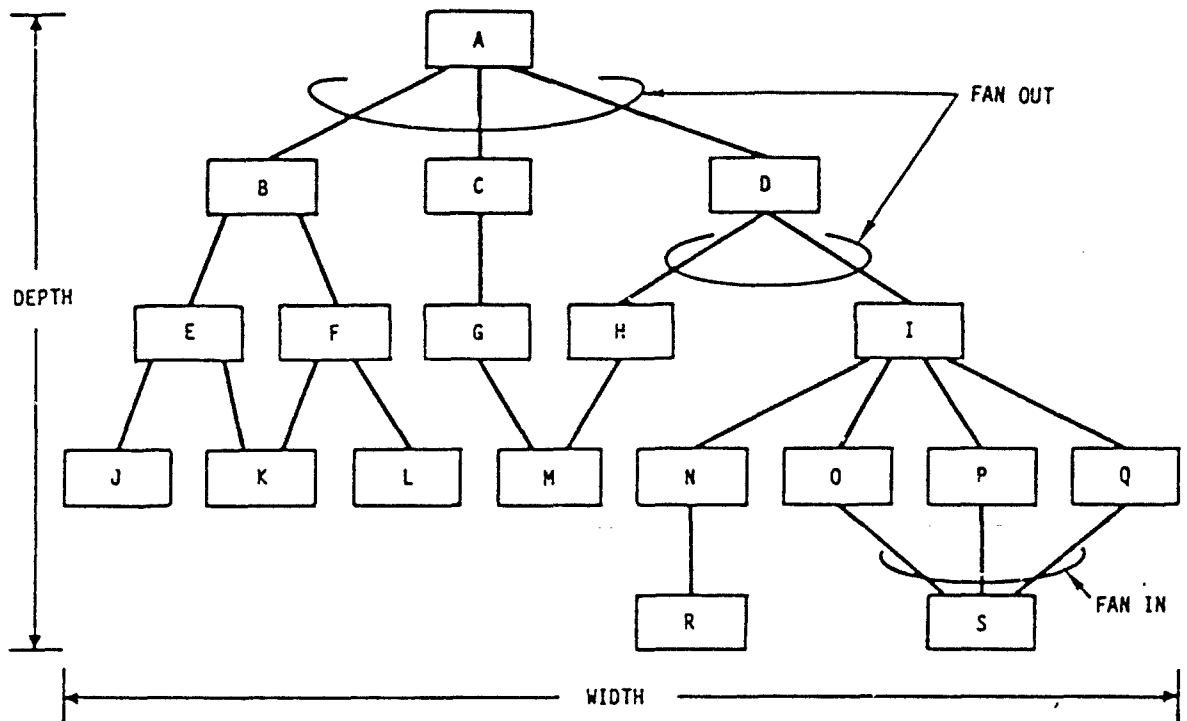


Figure 3. Measures of Structure

The control relationship among modules may be expressed in the following way. A module that controls another module is said to be superordinate to it. Conversely, a module controlled by another is said to be subordinate to the controller. For example, in Figure 3, module A is superordinate to modules B, C, and D and ultimately superordinate to all modules in the system. Module K is directly subordinate to modules E and F and ultimately subordinate to modules B and A. The depth of module B is 3 and the width of module I is 5.

### 3.0 MODULARITY

The concept of modularity in computer software has been espoused for two decades. Myers has stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program comprised of a single module) cannot be easily understood. The number of control paths, span of reference, number of variables, and overall complexity would make maintainability extremely difficult.

It has been shown in literature that partitioning a software solution into modules increases understanding and greatly enhances maintainability. It is easier to solve a complex problem if it is reduced to manageable pieces. However, this can be carried only so far. Other factors besides module size must be considered when developing software. As software is partitioned into modules, the requirements to interface information among the modules increases. We cannot simply make modules smaller and smaller and expect the total system complexity to decrease. When assessing software at USAEPG, the concepts of module cohesion (strength) and coupling must be considered. These concepts are described in Section 4 of this report.

Given the same set of requirements, the greater number of modules means a smaller module size and, therefore, less cost and effort to create and maintain each module. However, as noted above, the requirements to interface the modules increases. Figure 4 illustrates the relationship between modularity and software cost or effort (development or maintenance). There is a number of modules somewhere in the region between  $M_1$  and  $M_2$  that would result in minimum effort. However, we do not currently have the necessary sophistication to predict what this number should be.

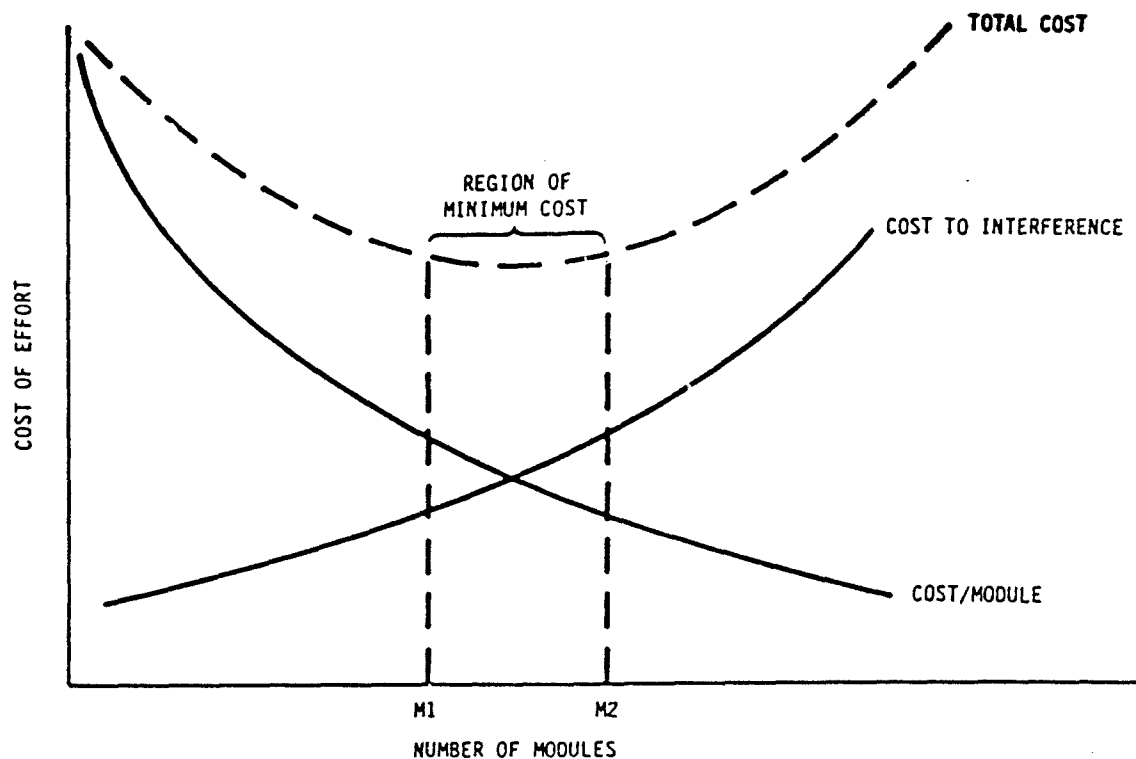


Figure 4. Modularity and Software Cost

In the sections that follow, some guidelines are presented to help determine how software should be modularized. However, before discussing coupling and cohesion, the concepts of abstraction, information hiding, and module type are presented.

### 3.1 Abstraction

When a modular solution to a problem is considered, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms, using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

Each step in the software engineering process is a refinement in the level of abstraction of the software solution. During system definition, software is described as a complete system element in the context of the entire system. During software planning and requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move from preliminary to detailed design, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

The concepts of stepwise refinement and modularity are closely aligned with abstraction. As the software design evolves, each level of modules in software structure represents a refinement in the level of abstraction of the software. In reality, a factored (Section 2) structure distributes levels of control and decision making, that is, levels of abstraction.

### 3.2 Information Hiding

The principle of information hiding suggests that modules be "characterized by design decisions that each hides from all others." In other words, modules should be specified and designed so that information (procedure and data) contained within a module are inaccessible to other modules that have no need for such information.

The term "hiding" implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information that is necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that comprise the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

The use of information hiding as a design criteria for modular systems provides greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are "hidden" from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

### 3.3 Module Types

Abstraction and information hiding are used to define modules within a software structure. Both of these attributes must be translated into module operational features that are characterized by time history of incorporation, activation mechanism, and pattern of control.

Time history of incorporation refers to the time at which a module is included within a source language description of the software. For example, a module defined as a compile time macro is included as in-line code by the compiler when an appropriate reference is made. A conventional subprogram (e.g., a subroutine or procedure) is included through generation of branch and link code.

Two activation mechanisms are encountered. Conventionally, a module is invoked by reference (e.g., a "call" statement). However, in real-time applications, a module may be invoked by interrupt; that is, an outside event causes a discontinuity in processing that results in passage of control to another module. Activation mechanics are important because they can affect software structure.

The pattern of control of a module describes the manner in which it is executed internally. Conventional modules have a single entry and exit and are executed sequentially as part of one user task. More sophisticated patterns of control are sometimes required. For example, a module may be reentrant. That is, a module is designed so that it does not in any way modify itself or the local addresses that it references. Therefore, the module may be used by more than one task concurrently.

Within a software structure, a module may be categorized as:

- o A sequential module that is referenced and executed without apparent interruption by the applications software.
- o An incremental module that can be interrupted prior to completion by applications software and subsequently restarted at the point of interruption.
- o A parallel module that executes simultaneously with another module in concurrent multiprocessor environments.

Sequential modules are most commonly encountered and are characterized by compile time macros and conventional subprograms -- subroutines, functions, or procedures. Incremental modules, often called coroutines, maintain an entry pointer that allows the module to restart at the point of interruption. Such modules are extremely useful in interrupt-driven systems. Parallel modules, sometimes called conroutines, are encountered when high-speed computation (e.g., pipeline processing) demands two or more CPUs working in parallel.

A typical control hierarchy (a factored structure) may not be encountered when coroutines or conroutines are used. Such nonhierarchical or homologous structures require special design approaches that are in early stages of development. Software assessment should include information with respect to atypical structures.

#### 4.0 MODULE INDEPENDENCE

The concept of module independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding. In landmark papers on software design, Parnas and Wirth allude to refinement techniques that enhance module independence. Later work by Stevens et al. solidified the concept.

Module independence is achieved by developing modules with "single-minded" functions and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific subfunction of requirements and has a simple interface when viewed from other parts of the software structure.

It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design and code modification are limited, error propagation is reduced, and "plug-in" modules are possible. To summarize, module independence is a key to good design, and design is the key to software quality and software quality test assessment.


Independence is measured using two qualitative criteria: cohesion and coupling. Cohesion is a measure of the relative functional strength of a module. Coupling is a measure of the relative interdependence among modules.

##### 4.1 Coupling

Coupling is defined as the degree of interdependence between two modules. The better the system, the lower its coupling. Coupling is concerned with how two modules communicate with each other whether it be by parameters, a global data area or by referring to data inside of the other. Low or loose coupling means that no module has to worry about the internal

workings of the other which makes for simple to understand systems. In addition, modules with low coupling have fewer connections meaning less chance for a bug in one module appearing as a symptom in another (ripple effect). They also minimize the risk of affecting one module due to changing another.

Five different types of coupling may occur between modules. They are:

- |                     |               |
|---------------------|---------------|
| 1. data coupling    | good or loose |
| 2. stamp coupling   |               |
| 3. control coupling |               |
| 4. common coupling  |               |
| 5. content coupling | bad or tight  |
- 

Two modules may be coupled by more than one type of coupling or by the same type a number of times. If two modules are coupled in more than one way, their coupling is defined by the worst (tightest) coupling they exhibit. The following sections describe these types.

#### 4.1.1 Data Coupling

Two modules are data coupled if they communicate by parameters, each parameter being a single variable, constant, or homogeneous table (a table in which each entity is of the same type). All that is passed between the modules is the necessary data it needs to carry out its function. For example, employee salary rate would not be passed to a module who's function is to print the employee's address on an envelope. This type of coupling which is unavoidable and perfectly acceptable, represents the most desirable type of coupling. Modules A and C of Figure 5 are examples of data coupled modules.



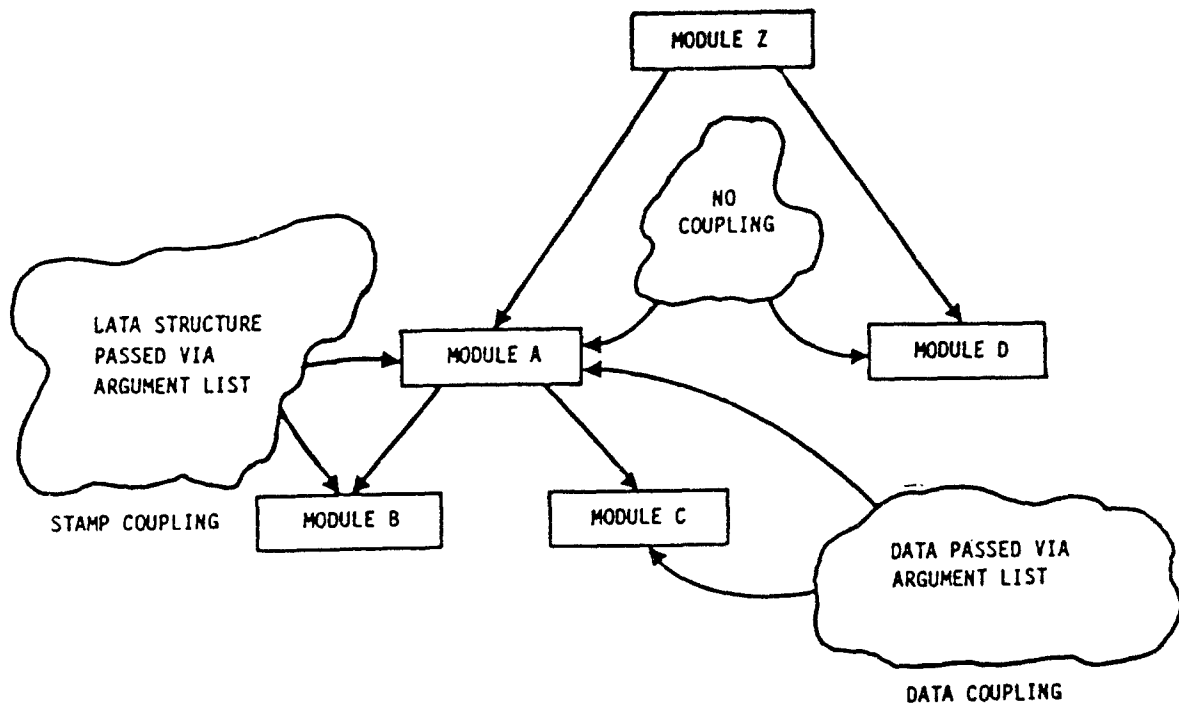


Figure 5. Low Coupling

#### 4.1.2 Stamp Coupling

A group of modules are stamp coupled if they pass a composite piece of data consisting of a number of fields. Data structures such as records are a good way of cutting down on excessive data coupling as long as all the fields have meaning for all modules concerned. However, stamp coupling should be avoided where possible because it creates unnecessary connections between modules. Suppose module B only needs a few fields in a record. By passing the entire record, B is forced to be aware of the entire structure of the record and B's chances of inadvertently modifying the record are increased. Figure 5 illustrates stamp coupling between modules A and B.

#### 4.1.3 Control Coupling

When one module passes to another, a piece of information intended to control the internal logic of the other, they are said to be control coupled. This piece of information can take the form of control flag or can be implied in a piece of data. This type of coupling is undesirable because the calling module must understand the logic of the subordinate module.

Thus, an overall understanding of each module becomes more difficult. Another undesirable situation that a passed control flag indicates is that a function has been split between two modules which should in fact be kept in one.

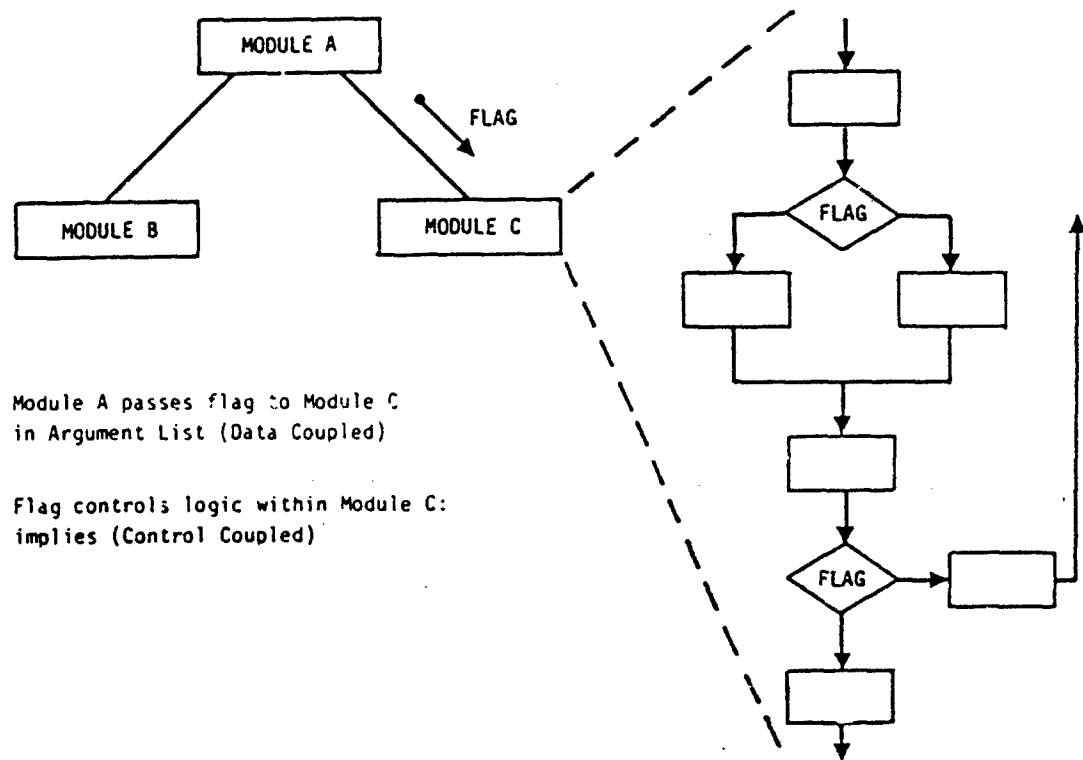


Figure 6. Moderate Coupling

#### 4.1.4 Common Coupling

Two modules are common coupled if they reference the same global data area. FORTRAN modules referencing data in a COMMON area and groups of modules referencing data in an absolute storage location (including registers) are examples of common coupling. The idea of modularity is degraded by common coupling because data is not confined to a single module or at most a small group of modules. Some problems which could arise because of common coupling are:

- A bug in any module using the global area may show up in any other module using the global area.
- Modules using global data areas use explicit names to reference the data, hence the module cannot be used for another application in which the variable names are different.
- Modules and the data it uses become more difficult to understand.
- If a piece of data or its structure within the global data area changes then all modules which reference that global area must be modified which is time consuming and, hence, less maintainable.

It should be noted that if two modules are common coupled, it is not necessary that one module call the other, just the fact that the two reference the same data area is enough criteria for common coupling (see Figure 7).

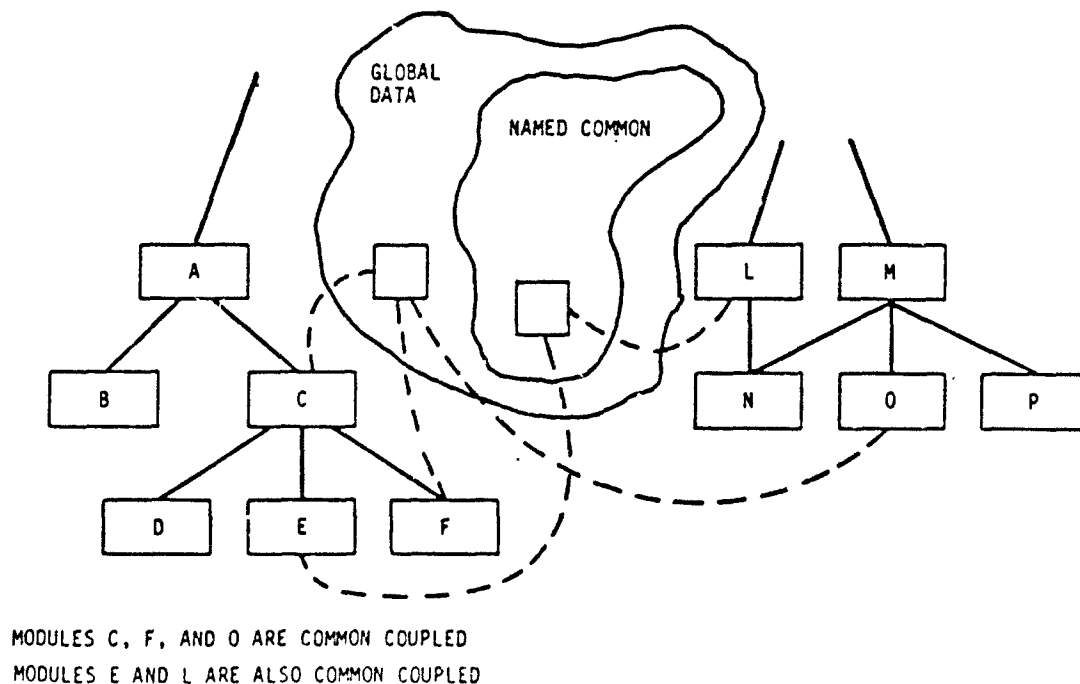


Figure 7. High Coupling

#### 4.1.5 Content Coupling

Content coupling pertains mostly to assembly language. It refers to the situation in which one module directly references the contents of another. For instance, if module A somehow references data in module B by using an absolute displacement, the modules are content coupled. Almost any change to B, or maybe just recompiling B with a different version of the compiler will introduce error into the program. Fortunately, most high order languages offer no way to implement content coupling.

#### 4.1.6 Determining Coupling Type

Data coupling between a calling and called module is recognized when the parameters of the call contain only simple data types or homogeneous tables. Stamp coupling is recognized when one or more parameters is a non-homogeneous composite piece of data such as a record. If one or more of the parameters passed is used in a condition statement within the called module, the two modules are control coupled. Common coupling can be detected by modules according to the global data areas they reference. The modules within each group are common coupled. If a module contains a branch to a label which is not defined in the scope of the module itself, it is content coupled with the module in which the label is defined.

Automatic determination of module coupling by a static analyzer can be achieved by examining data flow between modules and applying the above criteria to categorize the coupling type.

#### 4.1.7 Summary of Coupling

The following is a table which summarizes each type of coupling and their specific qualities.

<u>COUPLING TYPE</u>	<u>SUSCEPTIBILITY TO RIPPLE EFFECT</u>	<u>MODIFI- ABILITY</u>	<u>UNDER- STANDABILITY</u>	<u>MODULE'S USABILITY IN OTHER SYSTEMS</u>
Data	Variable	Good	Good	Good
Stamp	Variable	Medium	Medium	Medium
Control	Medium	Poor	Poor	Poor
Common	Poor	Medium	Bad	Bad
Content	Bad	Bad	Bad	Bad

#### 4.2 Cohesion

Modules are made up of elements -- pieces of code which accomplish some task. An instruction, a group of instructions, or a call to another module are such elements. Cohesion is a measure of the strength of functional associations of the elements within a module and therefore is synonymous with module strength. Strong, highly cohesive modules contain elements which are genuinely related. Their independence from information in other unrelated modules and their unity make for low coupling and easily maintained modules.

There are seven different degrees of cohesion which will be discussed in subsequent sections. These are in order of strength: functional, sequential, communicational, procedural, temporal, logical, and coincidental. The first three, functional, sequential, and communicational represent the most easily maintained modules, while the lower levels of cohesion, procedural, temporal, logical and coincidental are less easily maintained.

#### SCALE OF COHESION

<u>Type</u>	<u>Maintainability</u>	<u>Segmentation</u>
Functional	Best	Black box
Sequential Communicational		Not-quite-so- black box
Procedural Temporal		Gray box
Logical Coincidental	Worst	White or transparent box

#### 4.2.1 Functional Cohesion

Those modules which are functionally cohesive are the most easily maintained modules and have the lowest coupling. They contain elements which all contribute to one and only one problem-related task. The module may call other modules in order to solve subproblems of the main task and these may in turn call other modules, but the calling module is still considered functionally cohesive as long as it accomplishes one problem-related function. "Calculate Net Employee Salary" is an example of one such module with "Calculate FICA Deductions" a subfunction which it calls.

#### 4.2.2 Sequential Cohesion

The second most cohesive module is one that is sequentially cohesive. Its elements are involved in activities which are carried out in a specific order so that the result of one is the input data for the next. "Format and Cross-Validate Record" which has three activities, "format raw record", "crossvalidate fields in record", and "print error or confirmation message", is an example of a sequentially cohesive module. Each activity depends on the result of the previous activity. Like the functionally cohesive module, this type also has low coupling but unlike the former, its activities cannot be summed up as a single independent function and therefore is not as easily reusable by other programs.

#### 4.2.3 Communicational Cohesion

A communicationally cohesive module is one whose elements make up activities which use the same data. Like sequential cohesion, its activities cannot be summed up as one problem-solving function. In contrast to sequentially cohesive modules however, execution of communicationally cohesive modules activities do not have to be carried out in a specific order. These modules also maintain clean coupling but ease of maintainability is lost because attempting to modify one activity will usually affect another erroneously. Also if only one activity is really needed, a call to the communicationally cohesive module will result in unnecessary production of data and/or execution of code.

#### 4.2.4 Procedural Cohesion

Previous to this level of cohesion, a module's activities were strongly related to data. As we cross the boundary from easily maintainable modules to the less easily maintainable with procedural cohesion, it is control, not data, that flows from one activity to the next. A procedurally cohesive module contains elements which are involved with different and possibly unrelated activities. They are placed in one module because, although they may be unrelated, they must be carried out in some specific order.

An example is a module called "NEWTRAN" which contains two activities -- "update record on file" and "get next transaction". Usually much data must be passed to these types of modules leading to poor coupling.

#### 4.2.5 Temporal Cohesion

These modules have elements which are involved with activities that are related in time. Temporal cohesion is similar to communicational cohesion except that each activity must be carried out at the same time instead of on the same data. It is different from procedural cohesion because execution order is unimportant. The classic example of such a module is an initialization module. Coupling is fairly poor and the module is not easily reused.

#### 4.2.6 Logical Cohesion

A logically cohesive module contains elements that contribute to activities of the same general category. This type of cohesion differs from the previously defined types because not all of its activities will necessarily be carried out. The input data to the module determines execution which leads to tight coupling and possibly unused parameters.

Not only does this type of cohesion make for code which is difficult to understand, it also violates the principle of independent modules since an

outside source controls the inner workings of the module. An example of such a module is one which produces either a sales report, a project status report, or a customer transaction report, depending on whether the parameter flag is 1, 2 or 3.

#### 4.2.7 Coincidental Cohesion

Coincidental cohesion is similar to logical cohesion except that the activities of a coincidentally cohesive module do not even belong to a similar category as in logically cohesive modules. Organization is not based on either control or data flow but is dependent on an input flag to tell them what to do, similar to logical cohesion. It does not carry out one well-defined function and is difficult to understand. Such modules are rare and represent the lowest level of cohesion possible.

#### 4.2.8 Determining Module Cohesion

Figure 8 depicts a decision tree which determines the level of cohesion of a module. By answering the questions and following the paths indicated, one will eventually arrive at the level of cohesion for that module.

#### 4.2.9 Summary of Cohesion

The following table presents a summary of the specific qualities of each type of cohesion.

COHESION LEVEL	COUPLING	CLEANLINESS OF IMPL- MENTATION	USABILITY IN OTHER PROGRAMS	MODIFI- ABILITY	UNDERSTAND- ABILITY
Functional	Good	Good	Good	Good	Good
Sequential	Good	Good	Medium	Good	Good
Communicational	Medium	Good	Poor	Medium	Medium
Procedural	Variable	Medium	Poor	Variable	Variable
Temporal	Poor	Medium	Bad	Medium	Medium
Logical	Bad	Bad	Bad	Bad	Poor
Coincidental	Bad	Bad	Bad	Bad	Bad



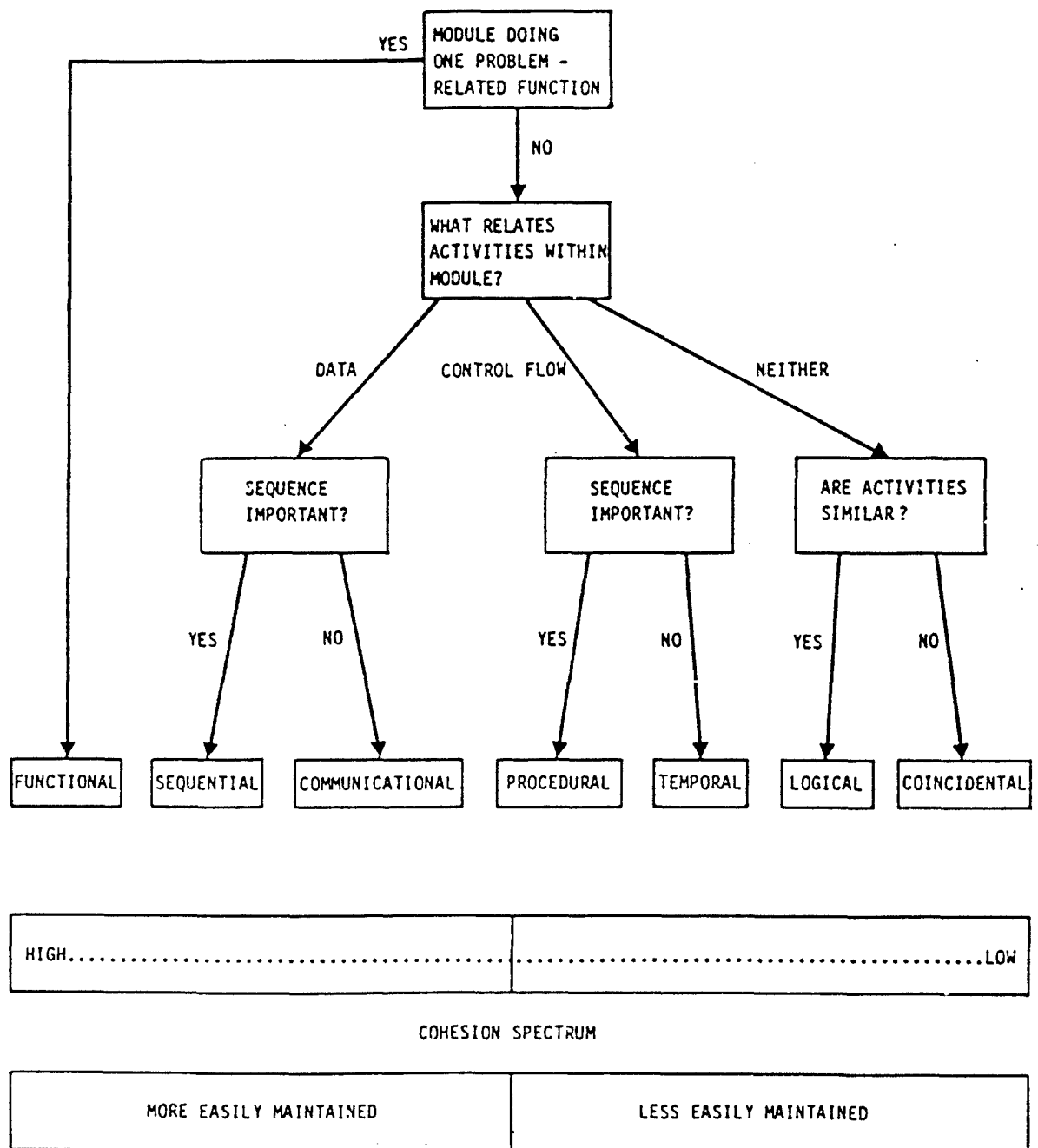


Figure 8. Determination of Cohesive Type

## 5.0

## REFERENCES

1. Myers, G., Reliable Software Through Composite Design, 1975
2. Page-Jones, Meilier, "The Practical Guide to Structured Systems Design", 1980
3. Parnas, D.L., "A Technique for Software Module Specification With Examples", Communications of the ACM, 1972
4. Rullo, Thomas A., "Advances in Computer Programming Management", 1980
5. Warnier, Jean-Dominique, Logical Construction of Systems, 1981
6. Yourden, E. and Constantine, L., Structured Design: Fundamentals of a Discipline of a Computer Program and Systems Design, 1978

APPENDIX J

MCCABE'S ESSENTIAL COMPLEXITY MEASURE



## TABLE OF CONTENTS

	PAGE
1.0 INTRODUCTION.....	1
2.0 OVERVIEW OF THE APPROACH.....	2
3.0 DETAILED DESCRIPTION OF THE METHOD.....	5
3.1 Depth-First Ordering of a Flow Graph .....	5
3.2 Dominators and Subordinates .....	6
3.3 The Subgraph Between Two Nodes .....	7
3.4 Branch at Exit Node .....	7
3.5 Illegal Branches Into the Subgraph .....	8
3.6 Illegal Branches Out of the Subgraph .....	8
3.7 Reduction of Complexity Measure .....	8
4.0 PDL FOR COMPLEXITY REDUCTION.....	9
5.0 REFERENCES.....	11

## REDUCTION OF PROGRAM COMPLEXITY: McCABE'S ESSENTIAL COMPLEXITY

### 1.0 INTRODUCTION

The complexity of the control structure of a module gives a good measurement of the complexity of the module. The control structure itself can be represented by its flow graph. Thus, by studying the flow graph, one can obtain the complexity measure of the module. This is the approach used by McCabe to compute program complexity as reported in the IEEE Transactions on Software Engineering, SE-2(4) 1976. This is referred to as the cyclomatic complexity of the module.

Oftentimes, the complexity of a program can be reduced by partitioning off appropriate portions of the program as subroutines or subprocedures. These portions correspond to the 'removable' subgraphs of the flow graph. Condensing each of these removable subgraphs into a single node reduces the graph into a simpler graph. McCabe defined the complexity of the reduced graph as the essential complexity of the program. It is the purpose of this report to present algorithms for identifying removable subgraphs of a flow graph and describe a method to compute McCabe's essential complexity.

The issue of identifying removable subgraphs is discussed below. First, an overview of the approach is presented. Secondly, the recommended method for determining the subgraphs is discussed in detail. Finally, a high level Program Design Language representation of the algorithm is presented.

## 2.0 OVERVIEW OF THE APPROACH

The removable subgraphs can be determined by successive elimination. First, a coarse criterion is used to find all subgraphs that are likely candidates for removal, and then apply a series of discriminatory criteria to rule out subgraphs that are not truly removable. The following is a summary of the procedure. An example is given at the end to illustrate the procedure:

Step 1. Every removable subgraph has a unique entry node and a unique exit node. The first step is to search through the flow graph to find all pairs of nodes that might constitute the entry node and the exit node of a removable subgraph. This is done by examining the dominator tree and the subordinate tree of the flow graph. The concept of dominator and subordinate is discussed in detail in Section 3.

Step 2. For each pair found in Step 1, determine the subgraph between the two nodes. The subgraph consists of nodes that are dominated by the entry node and have the exit node as a subordinate. Discard all subgraphs consisting of a single node.

The next 3 steps eliminate subgraphs with illegal branches. Moreover, Step 1 above rules out the following cases: branching out of the subgraph to a node that is not an ancestor of the entry node, and branching into the subgraph from a node that is not a descendant of the exit node from consideration.

Step 3. If the subgraph branches at the exit node to more than one outside node, eliminate the subgraph from consideration.

Step 4. Eliminate the subgraph if it can be entered from a descendant node of the exit node. This step takes care of backward branching into the subgraph.

Step 5. Eliminate the subgraph if there is an exit from the subgraph to an ancestor node of the entry node. This step takes care of backward branching out of the subgraph.

Step 6. Compute the reduced complexity measure. Let  $v$  be the complexity measure of the original flow graph. Suppose a removable subgraph is replaced by a node, then the complexity  $v$  is reduced by  $(e - n + 1)$ , where  $n$  is the number of the nodes in the subgraph and  $e$  is the number of edges. In other words, if  $\text{red}_v$  is the reduced complexity, then

$$\text{red}_v = v - (e - n + 1).$$

The flow graph shown in Figure 1 is used to illustrate the above procedure. This graph has 9 nodes, with node 1 as the initial node and node 9 as the terminal node.

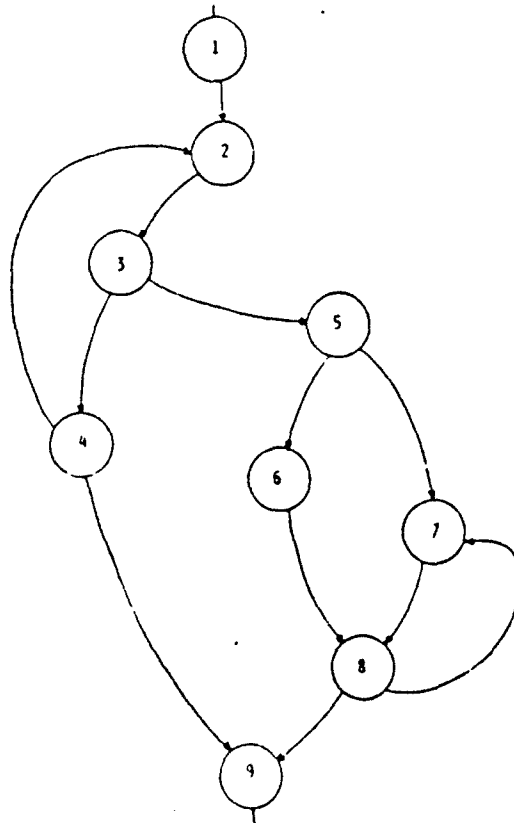


Figure 1. Flow Graph

Applying Step 1 to the above example produces the following pairs of nodes: (1,2), (1,3), (2,3), (2,9), (3,9) and (5,8). In addition, all pairs of the form (n,n) are also produced (it is possible that the entry node and exit node coincide, for example, a loop).

At Step 2, following subgraphs are produced: (<a,b> denotes the subgraph between node a and node b.)

- <1,2> : nodes 1 and 2
- <1,3> : nodes 1, 2 and 3
- <2,3> : nodes 2 and 3
- <2,9> : nodes 2, 3, 4, 5, 6, 7, 8 and 9
- <3,9> : nodes 3, 4, 5, 6, 7, 8 and 9
- <5,8> : nodes 5, 6, 7 and 8

Subgraphs of the form <n,n> are eliminated since they are all single node graphs. Let us make some comments on each of these subgraphs.

- <1,2> : there is a backward branch from node 4 into it
- <1,3> : branches to nodes 4 and 5 at the exit node (node 3)
- <2,3> : same as above
- <2,9> : qualified as a removable subgraph
- <3,9> : there is a backward branch out of the subgraph to node 2
- <5,8> : qualified as a removable subgraph

<1,3> and <2,3> are eliminated at Step 3, <1,2> is eliminated at Step 4 and <3,9> at Step 5.

Suppose that the subgraph <5,8> is replaced by a single node. This subgraph has 5 edges and 4 nodes. Therefore, the complexity of the procedure in Figure 1 is reduced by  $2 = 5 - 4 + 1$ .



### 3.0 DETAILED DESCRIPTION OF THE METHOD

In this section, the details of the approach outlined above are described. The concepts that are crucial to an algorithm are presented.

#### 3.1 Depth-First Ordering of a Flow Graph

Most of our algorithms involve searching the nodes of the flow graph. The way the nodes in a flow graph are labeled is quite arbitrary. However, when these nodes are ordered in the 'depthfirst' fashion, a lot of redundant search can be avoided. In this section, the depth-first search and the depth-first ordering of a flow graph is described.

The depth-first search starts at the initial node. At the beginning, all nodes are marked unvisited except the initial node. At each step, downward search from the current node is attempted. If there is any unvisited child node, the child node is marked visited and is chosen as the new current node, and the search proceeds as before. If the downward search is impossible (i.e., if the current node has no unvisited child nodes or no child nodes at all), the search returns to the parent node. The parent node becomes the current node and another downward search begins. The search stops when no downward search from the initial node is possible.

For example, consider the graph of Figure 1. The order in which the nodes are visited using depth-first search is as follows:

1, 2, 3, 4, 9, 4, 3, 5, 6, 8, 6, 5, 7, 5, 3, 2, 1

The search moves downwards first. It turns upwards at node 9. Downward search is attempted at node 4 and fails. The search goes back one more step to node 3 and turns downwards. It turns upwards again at node 8, downwards at node 5, upwards at node 7, and ends at node 1. The reverse of the order in which the nodes are last visited is called the depth-first ordering of the flow graph. The depth-first ordering of the above example is

1, 2, 3, 5, 7, 6, 8, 4, 9

A general algorithm for performing the depth-first search of a flow graph starting at an arbitrary node is given in Section 4.

### 3.2 Dominators and Subordinates

A discussion of the relationship between the entry node and the exit node of a removable subgraph is presented. The relationship may be characterized in terms of 'dominator' and 'subordinate' nodes.

Consider two nodes of a flow graph, say node  $i$  and node  $j$ . Node  $i$  is said to dominate node  $j$  if every path from the initial node to node  $j$  passes through node  $i$ . For example, in the flow graph of Figure 1, node 3 dominates node 8. On the other hand, node 8 is not dominated by node 6. The domination relation is transitive: if node  $i$  dominates node  $j$  and node  $j$  dominates node  $k$  then node  $i$  dominates node  $k$ . By definition, every node dominates itself, and is dominated by the initial node. In Section 4 we give an algorithm for constructing the table of dominators.

The opposite concept of domination is subordination. Node  $j$  is a subordinate of node  $i$  if every path from node  $i$  to the terminal node passes through node  $j$ . For the flow graph of Figure 1, node 8 is a subordinate of node 6, but not of node 3.

Consider a removable subgraph. Since it has a unique entry node, the entry node dominates every node in the subgraph. In particular, it dominates the exit node. Similarly, the exit node is a subordinate of the entry node. The first step of our procedure is to search all pairs  $\langle a, b \rangle$  and select those with the property that node  $a$  is a dominator of node  $b$  and node  $b$  is a subordinate of node  $a$ .

Take the flow graph of Figure 1 as an example. It can be seen that node 5 dominates node 8 and node 8 is a subordinate of node 5. Therefore  $\langle 5, 8 \rangle$  is a qualified pair. On the other hand, although node 3 dominates node 8, node 8 is not a subordinate of node 3. Therefore, we can disregard the pair  $\langle 3, 8 \rangle$ .

A separate algorithm for finding subordinates is not necessary. In fact, if we reverse all arrows in the original flow graph and consider the terminal node as the initial node and vice versa, then node  $j$  is a subordinate of node  $i$  in the original flow graph if and only if node  $j$  is a dominator of node  $i$  in the reversed graph. By applying the dominator algorithm to the reversed flow graph, the table of subordinates can be obtained.

### 3.3 The Subgraph Between Two Nodes

Suppose node  $a$  is a dominator of node  $b$  and node  $b$  is a subordinate of node  $a$ . Therefore, the portion of the flow graph consisting of the nodes that are dominated by node  $a$  and have node  $b$  as a subordinate as the subgraph between node  $a$  and node  $b$  may be removed. For simplicity, one can denote the subgraph by  $\langle a, b \rangle$  in the following. Using the tables of dominators and subordinates, one can determine the subgraph easily.

Consider the flow graph of Figure 1. As an example, the subgraphs  $\langle 5, 8 \rangle$  and  $\langle 2, 9 \rangle$  are determined.

Nodes that are dominated by node 5 are: 5, 6, 7 and 8.

Nodes of which node 8 is a subordinate are: 5, 6, 7 and 8.

Therefore  $\langle 5, 8 \rangle$  consists of nodes 5, 6, 7 and 8.

Nodes that are dominated by node 2 are: 2, 3, 4, 5, 6, 7, 8 and 9.

Node 9 is a subordinate of every node in the graph.

Therefore  $\langle 2, 9 \rangle$  consists of nodes 2, 3, 4, 5, 6, 7, 8 and 9.

### 3.4 Branch at Exit Node

By comparing the subgraph against the set of child nodes of the exit node, one determines whether the subgraph branches to more than one outside node when exiting. For example, consider the subgraph  $\langle 5, 8 \rangle$  in Figure 1. The subgraph consists of nodes 5, 6, 7, and 8. The exit node, node 8, has two child nodes: nodes 7 and 9. Since node 7 is in the subgraph, the subgraph exits to a unique node, namely node 9. On the other hand, subgraph  $\langle 1, 3 \rangle$  branches to nodes 4 and 5.

### 3.5 Illegal Branches Into the Subgraph

Suppose there is a branch into a subgraph  $\langle a, b \rangle$  from an outside node, say node  $c$ . Then node  $c$  is necessarily a descendant of node  $b$ . Otherwise there is a path from the initial node to node  $c$  to node  $b$  bypassing node  $a$ , i.e. node  $a$  does not dominate node  $b$ .

Therefore, to check that there are no illegal branches into a subgraph  $\langle a, b \rangle$ , one checks that none of the descendants of node  $b$  has a child node in  $\langle a, b \rangle$ . This can be done by looking up the ancestors-descendants table.

### 3.6 Illegal Branches Out of the Subgraph

Similarly, if there is a branch out of a subgraph  $\langle a, b \rangle$  to an outside node, this outside node must be an ancestor of node  $a$ . This is because node  $b$  is a subordinate of node  $a$ .

Again the illegal branches out of a subgraph  $\langle a, b \rangle$  can be detected by looking up the ancestor-descendant table, checking whether any of the ancestors of node  $a$  is a child node of some node in the subgraph.

### 3.7 Reduction of Complexity Measure

Suppose a removable subgraph is replaced by a node, then the number of edges in the original graph is reduced by the number of edges in the subgraph,  $e$ , and the number of nodes is reduced by  $(n - 1)$ , where  $n$  is the number of nodes in the subgraph. Since the complexity measure is defined as

$$(\# \text{ of edges}) - (\# \text{ of nodes}) + 2,$$

the complexity is reduced by  $e - n + 1$ , which is 1 less than the complexity of the removable subgraph.

#### 4.0 PDL FOR COMPLEXITY REDUCTION

Routine CMPX\_RED                   ... complexity measure reduction routine

.....  
.....  
.....

...       This routine accepts a flow graph as input and returns to the  
...       calling program the reduction in complexity when removable  
...       subgraphs are condensed into single nodes. The difference  
...       between McCabe's complexity measure and this return value is  
...       McCabe's essential complexity of the flow graph. The routine  
...       also outputs the list of the removed subgraphs together with  
...       their complexity measure.

...  
.....

...       generate data for later references

...       CALL GENO           .... generate descendant tree of the input flow graph  
...                           .... reverse input flow graph  
...                           .... reverse arrows in the graph  
...       CALL DFORD with input flow graph as argument  
...                           .... arrange nodes in depth-first order  
...       CALL DFORD with reversed graph as argument  
...                           .... arrange nodes in reverse depth-first order  
...       CALL DOMINATOR with input flow graph as argument  
...                           .... generate dominator tree of the flow graph  
...       CALL DOMINATOR with reversed graph as argument  
...                           .... generate subordinate tree of the flow graph

...  
...  
...

          initialization

clear CONDEMN\_LIST  
          .... this list accumulate nodes belonging to removed  
          .... subgraphs  
set RED\_IN\_CMPX to 0  
          .... no reduction in complexity

```

...
... Search for removable subgraphs. Pairs of nodes (node a, node b)
... are searched in the following fashion: node b in the reverse
... depth-first order, and for a fixed b, node a is searched in the
... depth-first order. If the subgraph between node a and node b
... is removable, the subgraph is listed in the output and is
... included in the CONDEMN_LIST. Nodes in the CONDEMN_LIST are
... skipped over in our search.
...
DO for each node, in the reversed depth-first order
  take this node as EXIT_NODE
  IF EXIT_NODE is not in the CONDEMN_LIST
  THEN
    .... get the entry node
    DO for each node, in the depth-first order
      take this node as ENTRY_NODE
      IF ENTRY_NODE is the initial node AND -
        EXIT_NODE is the terminal node
      THEN
        CYCLE .... not interested
      ENDIF
      IF ENTRY_NODE is not in the CONDEMN_LIST
      THEN
        IF ENTRY_NODE dominates EXIT_NODE AND -
          EXIT_NODE is subordinate to ENTRY_NODE
        THEN
          generate SUBGRAPH determined by two nodes
            .... consists of nodes dominated by
            .... ENTRY_NODE and have EXIT_NODE
            .... as a subordinate
          IF SUBGRAPH branches to more than one outside node
          THEN
            CYCLE .... subgraph not removable
          ENDIF
          IF there is a branch into SUBGRAPH from a -
            descendant node of EXIT_NODE
          THEN
            CYCLE .... subgraph not removable
          ENDIF
          IF SUBGRAPH branches to an ancestor node of -
            ENTRY_NODE
          THEN
            CYCLE .... subgraph not removable
          ENDIF
          compute cmpx(SUBGRAPH), the complexity of SUBGRAPH
          output SUBGRAPH and its complexity
          RED_IN_CMPX = RED_IN_CMPX + cmpx(SUBGRAPH) - 1
          include SUBGRAPH in CONDEMN_LIST
        ENDIF
      ENDIF
    ENDDO
  ENDIF
ENDDO
RETURN RED_IN_CMPX

```

## 5.0 REFERENCES

1. Matthew, S. Hecht, Flow Analysis of Computer Programs, Elsevier North-Holland Inc., New York, 1977
2. T. J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering (SE-2(4), Dec. 1976)

APPENDIX K

HALSTEAD'S SOFTWARE SCIENCE METRIC'S



## TABLE OF CONTENTS

	PAGE
1.0 INTRODUCTION.....	1
2.0 HALSTEAD'S METRICS.....	2
2.1 Program Length - N .....	2
2.2 Program Volume - V .....	3
2.3 Program Level - L .....	3
2.4 Potential Volume - V .....	3
2.5 Difficulty - D .....	3
2.6 Effort - E .....	3
3.0 PROGRAMMING APPROACH.....	4
4.0 DETAILED LOGIC.....	5
5.0 REFERENCES.....	6

## DETERMINATION OF HALSTEAD'S SOFTWARE SCIENCE METRICS

### 1.0 INTRODUCTION

Halstead theorized a method of analyzing computer language implemented algorithms by evaluating the static expressions that make up the algorithms. By identifying the operators and operands of the expressions and the number of times each is used, he defined measurable entities which serve to evaluate the complexity and quality of the algorithm's implementation. Furthermore these entities are general enough to be applicable to various computer languages. Some of these are:

- N - Program Length
- V - Program Volume
- L - Program Level
- V\* - Program Potential Volume
- D - Difficulty
- E - Effort

The next section explains in more detail the derivation and meaning of the above measures. Section 3 outlines one programming approach for gathering data for evaluation by Halstead's metrics. The fourth section contains brief PDL to assist in program creation.

## 2.0 HALSTEAD'S METRICS

All Halstead's metrics can be defined by determining the operators and operands used in the module under test and keeping track of how often they are used in executable statements of the module. Operators are defined as any symbol or keyword which specifies a specific algorithmic action. Operands are defined as any symbol which represents data.

Some examples of operators:

:=	;	goto	*
+	if	.and.	begin .. end

Some examples of operands:

SUM	500	1.73	"word"
-----	-----	------	--------

The following variables represent information obtained from the module under evaluation by monitoring operand and operator usage, which form the basis for determining Halstead's metrics.

n1 = number of unique operators occurring in the code  
n2 = number of unique operands  
N1 = total usage of all operators  
N2 = total usage of all operands

The next sections define Halstead's metrics in terms of the above variables.

### 2.1 Program Length - N

According to Halstead, the length of a program is the total number of times each operator and operand is used.

$$N = N1 + N2$$

## 2.2 Program Volume - V

This represents the size of the program in terms of bits. It is the length multiplied by the number of bits needed to encode the implementation's vocabulary. The vocabulary is the total number of distinct operators and operands appearing in the code.

$$V = N \log_2 (n_1 + n_2)$$

## 2.3 Program Level - L

The values L can take are  $\leq 1$ , with a value of 1 meaning that the program implements the algorithm in the most optimal and easily understood manner. It is a good indicator of a program's propensity for error and ease of understanding.

$$L = (2 \times n_2) / (n_1 \times N_2)$$

## 2.4 Potential Volume - V\*

The potential volume of a program represents the minimal and most concise form the algorithm for which it implements can take.

$$V^* = (L)(V)$$

## 2.5 Difficulty - D

Measures the average number of elementary mental discriminations required for each mental comparison needed to generate a program.

$$D = 1/L$$

## 2.6 Effort - E

E is the total number of elementary mental discriminations which were needed to generate a given program.

$$E = (V)(D) = (V)/(L)$$

### 3.0 PROGRAMMING APPROACH

The program to determine Halstead's complexity metrics shall be referred to as 'program analyzer' for the remainder of this text. Two sources of information will be required as input for the program analyzer. The first is the source code to be evaluated. The second is the set of operators particular to the language for which the program to be examined is written in. The program to be evaluated will be scanned module per module, line per line for operators and operands. Each module therefore, will have its own set of complexity metrics associated with it for which the program analyzer will determine.

Each line of code will be scanned and each symbol that makes up the line extracted. These symbols or 'tokens' are keywords, such as DO or IF, or identifiers, such as X or NUM, or operators such as < or +, and punctuation symbols such as commas or parenthesis. The following logic will apply to every token of the module to be analyzed:

If the token is an operator, it will be placed in an operator table only if that operator has not yet been entered. Also, an operator count variable will be incremented. If the token is an operand, it will be placed in an operand table only if that operator has not yet been entered. An operand count variable will also be incremented.

After the module has been completely scanned and all of its tokens examined,  $n_1$ ,  $n_2$ ,  $N_1$ , and  $N_2$  will be determined. The length of the operator table will define  $n_1$ . The length of the operand table will define  $n_2$ .  $N_1$  will be equal to the value of the operator count variable.  $N_2$  will be equal to the operand count value. Using these values, Halstead's various metrics will be computed and a report output.

#### 4.0 DETAILED LOGIC

```
Initialize_Tables(operator_tab, operand_tab : )
Read operators
optab_size = 0
opertab_size = 0
operator_count = 0
operand_count = 0
Dowhile not EOF
  Gettoken( : token)
  If token is an operator Then
    Begin
      If token not in operator_tab Then
        Begin
          optab_size = optab_size + 1
          operator_tab( optab_size ) = token
        End
      End
      operator_count = operator_count + 1
    End
  If token is an operand Then
    Begin
      If token not in operand_tab Then
        Begin
          opertab_size = opertab_size + 1
          operand_tab( opertab_size ) = token
        End
      End
      operand_count = operand_count + 1
    End
  Enddo
n1 = optab_size
n2 = opertab_size
N1 = operator_count
N2 = operand_count
Calculate_Metrics( n1, n2, N1, N2 : N, V, L, V*, D, E )
Print_Report(N, V, L, V*, D, E : )
End
```

## 5.0 REFERENCES

1. M. H. Halstead, Elements of Software Science, Elsevier North-Holland Inc., New York, 1977

APPENDIX L  
DISTRIBUTION



APPENDIX L. DISTRIBUTION LIST

<u>Addressee</u>	<u>Number of Copies</u>
Commander US Army Test and Evaluation Command ATTN: DRSTE-AD-M	3
DRSTE-10	2
DRSTE-AD-A	1
DRSTE-CT	3
DRSTE-CM	3
Aberdeen Proving Ground, MD 21005	
Commander Defense Technical Information Center ATTN: DTIC-CDR	2
Cameron Station Alexandria, VA 22314	
Commander US Army Aberdeen Proving Ground ATTN: STEAP-MT-M	2
Aberdeen Proving Ground, MD 21005	
Commander US Army Yuma Proving Ground ATTN: STEYP-MSA	
Yuma, AZ 85364	2
Commander US Army Jefferson Proving Ground ATTN: STEJP-TD-E	1
Madison, IN 47250	
Commander US Army Dugway Proving Ground ATTN: STEDP-PO-P	1
Dugway, UT 84022	
Commander US Army Cold Regions Test Center ATTN: STECR-TM	1
APC Seattle 98733	
Commander US Army Electronic Proving Ground ATTN: STEEP-MT-T	4
Fort Huachuca, AZ 85613	

Addressee

Number  
of Copies

Commander  
US Army Tropic Test Center  
ATTN: STETC-TD-AB  
APO Miami 34004

1

Commander  
US Army White Sands Missile Range  
ATTN: STEWS-AG-AS-AM (Record Copy)  
          STEWS-TE-PY  
White Sands Missile Range, NM 88002

1

4